# UNIVERSIDAD AUTONOMA DE MADRID

## ESCUELA POLITECNICA SUPERIOR



**Grado en Ingeniería Informática**

# TRABAJO FIN DE GRADO

**Artificial Composer: Optimizing Artificial Music**

**Alberto Prudencio de Dueñas**
**Tutor: Eduardo César Garrido Merchán**
**Ponente (si procede): Daniel Hernández Lobato**

**Junio 2018**

# Artificial Composer: Optimizing Artificial Music

**AUTOR: Alberto Prudencio de Dueñas**
**TUTOR: Eduardo César Garrido Merchán**

**Dpto. de Ingeniería Informática**
**Escuela Politécnica Superior**
**Universidad Autónoma de Madrid**
**Junio de 2018**

# Resumen

En este proyecto se desarrolla la creación de una herramienta que nos permita generar de forma pseudoaleatoria simulaciones musicales mediante técnicas de **composición artificial.**

Comenzaremos introduciendo el problema y la motivación detrás de este, así como los conceptos teóricos más importantes. Estos conceptos tratan los distintos modelos para la composición algorítmica y las metaheurísticas como métodos de optimización para problemas de carácter general. Para apoyar esto, se introducirán ejemplos de herramientas que usan modelos algorítmicos, así como algoritmos basados en metaheurísticas.

Una vez explicados los conceptos teóricos realizaremos una serie de objetivos, asunciones e hipótesis, y, mediante un diagrama de Gantt planificaremos el desarrollo del proyecto. Introduciremos las librerías que componen la herramienta, TinySound y Simulator, junto a sus características principales. Una vez explicadas por separada, abordaremos su integración y como gracias a esto obtenemos nuestro simulador.

Posteriormente, obtendremos resultados mediante distintos tipos de ejecución, y, analizaremos estos mediante la utilización de tablas y gráficas marginales. Estos resultados mostrarán las características de las simulaciones generadas, por lo que, podremos observar las diferencias entre aquellas simulaciones que no han sido optimizadas frente a las que sí, dándonos pie a la discusión sobre el grado de utilidad de la metaheurística implementada en el proyecto.

Finalmente, con el análisis de los resultados y las conclusiones obtenidas haremos un repaso de los objetivos, asunciones e hipótesis que realizamos previamente, discutiendo si se han cumplido o no y por qué. Junto a las conclusiones se incluirán una serie de características que serán implementadas al proyecto en un trabajo futuro.

# Palabras clave

Composición artificial, simulador, metaheurística, optimización, BPM (Beats por minuto) o tempo, kick, snare, clap, hihatsclosed, hihatsopen, cymbals, percussion, drum, TinySound, Recocido simulado, beat, bar.

# Abstract

In this project we are going to develope the creation of a tool that allows us to generate pseudo-random musical simulations using artificial composition techniques.

We will begin by introducing the problem and the motivation behind it, as well as the most important theoretical concepts. These concepts are based on the different models for algorithmic composition and metaheuristics as methods of optimization for general problems. To support this, examples of tools using algorithmic models will be introduced, as well as algorithms based on metaheuristics.

Once the theoretical concepts have been explained, we will carry out a series of objectives, assumptions and hypotheses and, thanks to a Gantt diagram, we will plan the development of the project. We will introduce the libraries that make up the tool, TinySound and Simulator, along with their main features. Once explained separately, we will deal with their integration and how thanks to this we obtain our simulator.

Subsequently, we will obtain results through different types of execution, and we will analyze these through the use of tables and marginal graphs. These results will show the features of the simulations generated, so we can see the differences between those simulations that have not been optimized against those that have, giving rise to discussion on the degree of usefulness of the metaheuristics implemented in the project.

Finally, with the analysis of the results and the conclusions obtained, we will review the objectives, assumptions and hypotheses that we previously carried out, discussing whether or not they have been met and why. Along with the conclusions will be included a series of characteristics that will be implemented to the project in future work.

# Keywords

Artificial composition, simulator, metaheuristic, optimization, BPM (Beats per minute) or tempo, kick, snare, clap, hihatsclosed, hihatsopen, cymbals, percussion, drum, TinySound, Simulated Annealing, beat, bar.

## *Agradecimientos*

La mayor inspiración que he tenido durante todo el trabajo han sido las enormes ganas que actualmente tengo de labrarme un futuro aunando mis dos mayores aficiones, informática y música.

Sin embargo, todo esto no habría sido posible sin el apoyo incondicional de mis padres, que siempre están ahí para lo que sea. Muchas gracias papá y mamá.

A mis hermanos, que, a pesar de pasar poco tiempo con ellos, para mí, son un ejemplo a seguir. Me habéis cuidado toda la vida como si fuera un hijo vuestro y no podría estaros más agradecido de todo lo que me habéis enseñado. Soy quién soy gracias a vosotros, y estoy muy orgulloso de ello.

A mi novia y mis amigos, que siempre están atentos a cómo van las cosas y dispuestos a echar una mano como sea. Muchas gracias por estar pendiente de mí y de haber tirado de mí en situaciones difíciles. Vosotros habéis influido mucho en mi ánimo, y gracias a ello, hemos podido terminar.

Por último, mencionar la excelente relación mantenida con el tutor. Desde el primer momento hemos sabido compenetrarnos y siempre que he necesitado ayuda, he podido acudir a él. Muchas gracias Eduardo.

# INDICE DE CONTENIDOS

# INDICE DE FIGURAS

# INDICE DE TABLAS



# INDICE DE GRÁFICAS

# 1 Introducción

## *1.1 Motivación*

La música es un campo que siempre está evolucionando, nuevos géneros, nuevos sonidos, nuevos instrumentos... Esta evolución es una consecuencia de que cada día podemos conocer mejor el sonido y experimentar con este fácilmente gracias a las altas prestaciones tecnológicas de las que disponemos hoy en día.

Uno de los campos que más se ha aprovechado de esta evolución es el de la música electrónica, es decir, aquella que ha sido producida mediante instrumentos y tecnología electrónica [**1**]. Como antes hemos mencionado, esto es posible gracias a los avances de los que disponemos, por lo tanto, podríamos decir que la experimentación con la música está muy ligada a la informática. Esta experimentación es realizada mediante software, y, está enfocada principalmente a la generación de música mediante redes neuronales profundas. Un ejemplo de esta aplicación es **Magenta**, un software de generación de música y arte que usa las técnicas mencionadas. Sin embargo, aquello en lo que nosotros nos centraremos, la **composición artificial**, se basa en la creación de composiciones musicales pseudo-aleatorias.

Las metaheurísticas [**2**] son unos métodos heurísticos utilizados para resolver problemas de carácter computacional general, es decir, aquellos problemas que no disponen de un algoritmo que proporcione una solución satisfactoria. Ejemplos de esto son, el **Recocido Simulado** [**3**] o la **Búsqueda Tabú** [**4**], algoritmos de búsqueda que se encargan de encontrar el valor óptimo dentro de un espacio determinado. Ambos realizan esto mediante la búsqueda de posibles soluciones mejores en estados vecinos, es decir, aquellos que forman parte del mismo espacio de búsqueda, pero se diferencian del estado actual en alguna/s características.

Dada la naturaleza aleatoria de las simulaciones que generamos, no podremos establecer un algoritmo concreto que proporcione simulaciones satisfactorias, sin embargo, podremos aplicar a las simulaciones un criterio que las clasifique por los atributos y parámetros que las definen. Si a esto añadimos la utilización de una metaheurística que nos permita optimizar nuestra función de evaluación, obtendremos finalmente unas simulaciones más complejas y bien definidas.

## *1.2 Objetivos*

Para el desarrollo de este trabajo vamos a establecer dos objetivos finales:

**1)** La creación de un simulador que genere de forma pseudoaleatoria ritmos musicales (*simulaciones*), los cuales estarán compuestos por sonidos que se obtendrán aleatoriamente de una biblioteca creada previamente para este propósito. Esta contará con los elementos principales de una batería, bombo, tambor, platillos…

   Llamamos a este simulador pseudoaleatorio ya que ciertas características de la simulación serán elegidas aleatoriamente entre un rango de valores predefinidos. Por ejemplo, si elegimos un bombo, aleatoriamente se elegirá el tiempo en el que este será reproducido dentro de una lista con marcas de tiempo.

**2)** La correcta evaluación y posterior optimización de las piezas musicales generadas por el simulador. Esto será llevado a cabo mediante la creación de una función de evaluación automatizable que evalúe los aspectos generales de las simulaciones junto a, un sistema de optimización basado en una metaheurística que nos permita obtener soluciones óptimas.

Sin embargo, para poder llevar a cabo estos objetivos debemos cumplir los siguientes objetivos primero:

**1)** Creación del simulador y del componente que se ocupará de la aleatorización de los sonidos utilizados y sus atributos.

**2)** Creación de un criterio automático de evaluación neutral para las piezas generadas del simulador.

**3)** Creación de la herramienta encargada de la optimización de las simulaciones.

**4)** Unión de estos elementos para la creación de un software que cree, evalúe y optimice simulaciones.

## *1.3 Organización de la memoria*

La memoria consta de los siguientes capítulos:

- **Introducción:** Introducción del trabajo en la que se habla acerca de la motivación que envuelve a este y de los objetivos a llevar a cabo.

- **Estado del arte:** Se explicarán conceptos relativos al proyecto de forma teórica y se mostrarán casos de uso de estos en la actualidad.

- **Diseño y desarrollo:** Se tratará el planteamiento del problema y de las distintas herramientas que se utilizarán, así como la relación existente entre estas junto a, una explicación de la metodología usada para llevar a cabo el desarrollo del problema y de las distintas herramientas a usar, así como todas las decisiones llevadas a cabo a lo largo de este.

- **Experimentos y resultados:** Realización de pruebas tanto unitarias como del conjunto final; análisis y evaluación de los resultados obtenidos.

- **Conclusiones y trabajo futuro:** Se tratarán las conclusiones obtenidas y se expondrá una reflexión acerca del trabajo futuro a realizar para la mejora del software desarrollado.

# 2 Estado del arte

A lo largo de este capítulo trataremos de una forma más teórica los conceptos básicos que envuelven el proyecto. Para ello, hablaremos sobre la simulación musical mediante composición artificial, así como las metaheurísticas como método de optimización.

## *2.1 Simulación musical mediante composición artificial*

La composición artificial está basada en la composición algorítmica [**5**], una técnica que ha sido usada desde hace siglos. Esta técnica consiste en la utilización de un algoritmo para componer piezas musicales. Un ejemplo de esto es el **Musikalisches Würfelspiel**, o juego de dados musical [**6**], donde se atribuían distintos pasajes de música a las caras de un dado, que, posteriormente sería lanzado y, según se obtuviesen los resultados, se construiría la pieza, poniendo una tirada detrás de otra.

Podemos diferenciar entre dos tipos de composición algorítmica:

**1)** La pieza ha sido enteramente compuesta y producida por un ordenador.
**2)** La pieza ha sido compuesta mediante una computadora y la intervención de un humano.

Para considerar que la pieza pertenece al tipo **1)**, el algoritmo que la ha compuesto debe tener la capacidad de tomar decisiones en la creación de esta. Sin embargo, este método es muy general, y no consigue representar las diferencias entre los algoritmos existentes, por lo que, debemos atender a la estructura y la manera de procesar los datos del algoritmo.

### 2.1.1 Modelos algorítmicos

Existen distintos tipos de modelos algorítmicos [**5**], por lo que, explicaremos cada uno de ellos:

- **Modelos matemáticos**: Basados en ecuaciones matemáticas y eventos aleatorios. La mayoría de composiciones creadas a partir de estos modelos están basadas en procesos estocásticos [**7**], es decir, métodos no deterministas, donde el compositor establecía las ponderaciones de los eventos aleatorios.

- **Sistemas basados en el conocimiento**: Fundamentados en la creación de piezas mediante el conocimiento de un determinado género musical. Con un conjunto de argumentos permite la creación de piezas que se asemejen al género conocido.

- **Gramáticas**: Creación de una gramática para tratar la música como una lengua. Las composiciones son creadas mediante una construcción gramatical que posteriormente es convertida en una pieza musical comprensible. Estas gramáticas pueden incluir reglas con el fin de realizar composiciones a gran escala. Encontramos más información sobre estas en [**8**].

- **Método evolucionario**: Utilización de algoritmos genéticos [**9**] y sus técnicas de mutación y selección natural con el fin de obtener una composición musical adecuada tras la acción iterativa del algoritmo. Las malas soluciones son eliminadas, mientras que las buenas sobreviven y se utilizan en próximas iteraciones.

- **Sistemas de auto-aprendizaje**: El programa aprende por él mismo mediante ejemplos proporcionados por el usuario/programador y genera sus propias composiciones en base a la información que ha recolectado de los ejemplos. Las composiciones generadas mantienen ciertas características de los ejemplos y son similares a estos. Varios ejemplos de esto en [**10**]

- **Sistemas híbridos**: Utilización de distintos métodos de forma conjunta con el fin de juntar todas las ventajas y eliminar los puntos débiles de cada algoritmo.

### 2.1.2 Herramientas basadas en la composición artificial

En este subapartado mencionaremos dos herramientas de las más desarrolladas en este ámbito que están basadas en la composición artificial:

- **Orb Composer** [**11**]: Orb composer es una herramienta inteligente y creativa desarrollada por Hexachords, que actúa como asistente para compositores musicales.

  Esta herramienta actúa mediante la creación de bloques a los cuales podemos asignarles unos parámetros (*intensidad, momento y espacio)*. Una vez creado el bloque, podremos convertirlo en una melodía, un clip de instrumentos, acordes, en notas, o en el conjunto de todos estos. Según el tipo de bloque elegido se generarán unos elementos u otros, pudiendo cambiar entre estos simplemente eligiendo un nuevo tipo de bloque. Cada vez que se elige un nuevo tipo para un bloque ya creado, se altera la música ya creada generando nuevas variaciones. Además de ser un software independiente, permite la conectividad en tiempo real con otros **DAW** [**12**].

- **Amper Music** [**13**]: Amper Music es una herramienta web gratuita desarrollada por un equipo de directores de cine y productores musicales de Hollywood con el objetivo de crear melodías y canciones de forma casi instantánea y que no tengan derechos de autor.

  De forma similar a Orb Composer, podemos crear bloques a los que, en este caso, les podemos asignar un género y estilo, distintos conjuntos de instrumentación, un BPM, la duración y la clave de la composición. Con estos parámetros elegidos, se generará una canción única, compleja y bien definida.

  Dispone de dos métodos de utilización, el simple, en el cual solamente le indicamos el género y estilo, y el pro, donde indicamos todos los parámetros posibles y se nos permite editar a nuestro antojo la canción generada.

## *2.2 Metaheurísticas como método de optimización*

Las metaheurísticas son procedimientos de búsqueda en los que "se parte de una solución inicial a la que se realizan modificaciones en sucesivas iteraciones para obtener una solución final" (García, s.f, p.2). Estas iteraciones proporcionan soluciones alternativas, conocidas como, soluciones vecinas, las cuales serán nuevas candidatas a ser la solución.

A diferencia de las heurísticas, las metaheurísticas son independientes del problema y huyen de óptimos locales. Esto nos permite resolver problemas de carácter general en los que no existe un procedimiento concreto que proporcione el resultado satisfactorio. Por lo que, si tratamos de optimizar un problema en el que los elementos que componen las soluciones se han obtenido de forma aleatoria es necesario utilizar una metaheurística como método de optimización.

Todas las técnicas metaheurísticas tienen las siguientes características:

- No saben si han alcanzado la solución óptima, por lo que necesitan una condición de parada.

- No garantizan la solución óptima ya que son algoritmos aproximativos.

- Ciertas iteraciones admiten soluciones peores a la actual con el fin de explorar el espacio de búsqueda más a fondo.

- Son sencillas, solamente necesitan una solución inicial, un espacio de búsqueda y un método para explorar este.

- Son aplicables a cualquier problema de optimización combinatoria. A pesar de esto, la eficiencia depende de si las operaciones que se realizan tienen relación con el problema en cuestión.

- La regla de selección es dependiente del instante en el que nos encontremos y del historial de soluciones anteriores, por lo que, a pesar de que exista una misma solución repetidas iteraciones, esta puede cambiar.

Esto es un ejemplo de una cita que citó García (como se cita en García, s.f, p.3) (Sadiq, S. M. y Habib, Y., 1999)

Como ya hemos mencionado, las metaheurísticas no proporcionan soluciones óptimas, si no, aproximaciones a esta, sin embargo, su independencia del problema y facilidad de implementación permiten el estudio de problemas de alta complejidad.

Existen diferentes técnicas de optimización mediante metaheurísticas, sin embargo, explicaremos en profundidad aquella que utilizaremos posteriormente en el desarrollo de nuestro problema, mientras que otras serán mencionadas brevemente.

### 2.2.1 Algoritmo de recocido simulado (Simulated Annealing)

Basado en la solidificación de los cuerpos sólidos, el algoritmo de **recocido simulado** recrea los cambios en la configuración que sufren estos durante este proceso. Cada configuración por la que pasa un sólido representa una solución posible al problema, por lo que, la solución, a medida que descienda la temperatura pasará por distintos estados, alcanzando finalmente un mínimo aproximado.

El proceso que sigue este algoritmo parte de una solución y temperatura inicial. De forma iterativa, se exploran distintas soluciones, llamadas soluciones vecinas y se disminuye la temperatura.

La probabilidad de aceptar una solución vecina para que sustituya a la actual es:

$$Prob\ aceptación = \begin{cases} 1 & si \quad f(i) \leq f(j) \\ \exp\left(\frac{-(f(j) - f(i))}{temp}\right) & si \quad f(i) > f(j) \end{cases}$$

Donde *f(j) – f(i)* representa la variación de energía entre la solución candidata y la solución actual, mientras que temp representa la temperatura actual del proceso.

Para el correcto funcionamiento del algoritmo es necesario definir las variables y condiciones que va a seguir este durante su ejecución, por lo que, es necesario establecer:

- **Solución inicial**: Estado inicial a partir del cual se realizarán modificaciones. Puede ser obtenido a partir de otros métodos, de forma aleatoria o de otra solución anterior.

- **Temperatura inicial**: Temperatura a la que comenzará el algoritmo. Preferiblemente alta para dotar al algoritmo de más iteraciones en las que explorar el espacio de búsqueda.

- **Soluciones vecinas**: Método de obtención/generación de soluciones vecinas a partir de la solución actual, así como el método de selección.

- **Ratio de enfriamiento**: Establecimiento de un valor constante que disminuirá la temperatura en cada iteración.

- **Temperatura final**: Idealmente 0, sin embargo, puede dar problemas a la hora de calcular las probabilidades de aceptación, por lo que es necesario establecer otro valor. Lundy y Mees (1986) proporcionan la siguiente fórmula:

$$T_f < \frac{\varepsilon}{\ln(n-1) - \ln(\theta) + \ln(1+\ \theta)} \approx \frac{\varepsilon}{\ln(n)}$$

  Donde θ es la probabilidad de que se obtenga una solución cuyo valor de la función objetivo menos el de la óptima global es menor que ε y n es el número de elementos del espacio de soluciones.

### 2.2.2 Algoritmos genéticos (Genetic algorithms)

Los algoritmos genéticos [**14**] se basan en "la mecánica de selección natural y de la genética natural" (Goldberg, 1989) para obtener de un conjunto inicial de soluciones, llamado población, los individuos que mejor se han adaptado a los cambios, es decir, los más fuertes.

Esto se consigue según Goldberg [**15**] gracias a que:

"Combinan la supervivencia del más apto entre estructuras de secuencias con un intercambio de información estructurado, aunque aleatorizado, para constituir así un algoritmo de búsqueda que tenga algo de las genialidades de las búsquedas humanas" (Goldberg, 1989)

Estos algoritmos funcionan mediante generaciones en las que las poblaciones evolucionan constantemente mediante los métodos de selección, cruce y mutación. Estos métodos actúan

sobre los valores que definen las soluciones, de forma que, al aplicar estos se generarán nuevas soluciones. Tras cada generación se eligen los individuos más fuertes (óptimos), y, con el paso de estas, las poblaciones existentes estarán compuestas cada vez por individuos más fuertes, es decir, mejores soluciones.

### 2.2.3 Algoritmo de la colonia de hormigas (Ant colony)

Basado en el comportamiento de las colonias de hormigas, este algoritmo [**16**], utiliza la probabilidad para resolver problemas de grafos o rutas.

Las hormigas al recoger comida y llevarla a la colonia dejan tras de sí un rastro de feromonas, el cual, ayuda a otras hormigas a volver a la colonia. De esta manera, cada vez que una hormiga deba volver a la colonia y tenga que escoger entre un camino u otro, aplicará probabilidades según los rastros de feromonas que encuentre. Finalmente, los caminos con mayor rastro de feromonas persistirán, mientras que, aquellos con menor, terminarán evaporándose.

Aplicando esto a problemas de optimización, cada hormiga construye una solución posible, por lo que, cada vez que una hormiga construye una solución, altera los valores de su ruta (feromonas) para que próximas hormigas puedan utilizarla en sus búsquedas. La evaporación de feromonas ayuda a reducir estos valores para evitar caer en óptimos locales.

De esta manera, todas las hormigas obtendrán una ruta óptima a la hora de llevar la comida a la colonia, es decir, construirán una solución satisfactoria.

### 2.2.4 Busqueda Tabú (Tabu Search)

La búsqueda tabú [**4**] "guía un procedimiento heurístico de búsqueda local en la búsqueda de optimalidad global" (Melián, s.f., p.1).

Al igual que otros métodos de optimización, se basa en la búsqueda de soluciones mediante soluciones vecinas, sin embargo, se diferencia de otras búsquedas principalmente en la utilización de una estructura de memoria que le permite almacenar soluciones que ya ha explorado con anterioridad con el fin de no volver a explorarlas, y así, explorar nuevas soluciones.

Estas soluciones que no vuelven a explorarse se almacenan en la **lista tabú.** Esta lista no es permanente, es decir, las soluciones que se almacenan estarán en la lista durante N iteraciones. N representa el tenor tabú (tabu tenure). Este valor puede ser el mismo para todos los elementos o ser variable.

El pseudocódigo de todos los algoritmos mencionados anteriormente se encuentra en el anexo **A.**

# 3 Diseño y desarrollo del proyecto

En este capítulo introduciremos la definición del proyecto, es decir, recapitularemos los objetivos e introduciremos las asunciones e hipótesis del proyecto, así como el alcance de este. Posteriormente explicaremos las librerías utilizadas y desarrolladas, con sus conceptos y utilidades de forma general, además de explicar de forma detallada los módulos y funciones principales que comprenden nuestro software.

## *3.1 Definición del proyecto*

Como ya hemos mencionado en la sección **1.2**, nuestros objetivos principales son dos:

- La creación de un simulador que, mediante una biblioteca de sonidos creada por el usuario, cree de forma pseudoaleatoria simulaciones que representen una composición musical compuesta por elementos de batería. Dentro de la aleatoriedad de este simulador, se permitirá elegir ciertos parámetros, como el BPM (Beats per minute) o el número de sonidos, los cuales serán explicados posteriormente.

- El desarrollo de una función de evaluación automatizable, es decir, que el propio simulador pueda realizar sin ayuda del usuario, así como, la utilización de un método de optimización matemática que nos permita maximizar esta evaluación, y, de esta manera generar simulaciones óptimas.

Siendo estos nuestros objetivos, podemos realizar una serie de asunciones, hipótesis y restricciones sobre el resultado final de nuestro simulador.

Damos por hecho que:

- Los sonidos en cada simulación serán diferentes gracias a la componente pseudoaleatoria del simulador.

- La función de evaluación tendrá una forma suave, es decir, si los parámetros sufren cambios mínimos, la evaluación también, y, si sufren cambios grandes, la evaluación cambiará drásticamente.

- La función de evaluación será barata de evaluar, ya que se tratará de una evaluación general sobre la simulación, con el fin de poder automatizar esta.

- La función de evaluación no proporcionará los mismos resultados para los mismos parámetros, es decir, tendrá ruido.

- La optimización escogida nos permitirá conseguir resultados mucho mayores respecto a la función de evaluación.

Suponemos que:

- Las simulaciones con mayor evaluación corresponden a ritmos más completos y bien definidos, es decir, aquellos que se ajustarían mejor a una canción.

- No todas las evaluaciones con valores altos deben ser buenas, o, aquellas con valores bajos malas.

- La optimización mediante una metaheurística es el método correcto para maximizar los valores obtenidos en la evaluación.

- El número de parámetros se limitará a los valores necesarios para parametrizar una evaluación.

- El proceso de creación de una canción a partir de una simulación, o la integración de esta en un software de audio serán tareas posteriores a la realización del TFG.

*Estas últimas dos suposiciones se deben a las limitaciones de tiempo del que se dispone para la realización del TFG.*

Para mostrar de una forma más visual la planificación establecida para el desarrollo del proyecto, y el alcance de este, se mostrará un diagrama de Gantt [**17**] sencillo. En este podremos observar el tiempo establecido para cada tarea y las fechas estimadas de inicio y fín de cada una. *Debido al tamaño del diagrama, se mostrará en el anexo **C.***

Teniendo como fecha límite el día 22 de junio, hemos conseguido planificar todo el proyecto para el día 13 de junio, lo que nos proporciona días extra en caso de que ocurriese cualquier contrariedad.

Todas las fechas son estimaciones realizadas respecto al papel que desempeña cada elemento, por lo que cosas como la clase principal del simulador, **simulationConfigurator**, de la que hablaremos más tarde, o las pruebas finales, que desempeñan papeles fundamentales, ocupan un mayor tiempo de desarrollo.

Para conocer a fondo el papel que desempeñan los elementos del diagrama, introduciremos las librerías **TinySound** y **Simulator**, y, por último, el método de optimización elegido.

## *3.2 TinySound*

**TinySound** es un sistema de sonido sencillo, pequeño y intuitivo desarrollado por Finn Kuusisto que está basado en las librerías de sonido de java y nos permite trabajar fácilmente con ficheros de audio.

Esta librería permite trabajar con muchos formatos de audio, así como con formatos de almacenamiento que la versión de java que usemos permita, además de permitir el formato Vorbis en los audios OGG gracias a las librerías JOrbis y VorbisSPI, que se ocupan de la decodificación de estos formatos, y de la inclusión de un soporte del formato Vorbis para Java respectivamente. Para esto, trabaja con una tercera librería llamada Tritonus Share, la cual es una implementación de la API de sonido de Java. A pesar de la gran cantidad de

formatos que puede soportar, ciertas codificaciones no son capaces de resolverse, y, por lo tanto, no podrán ser usadas por la librería.

Todos los audios son guardados con el formato de datos LPCM (Linear Pulse Code Modulation), de 16-bit, 44.1kHz y 2 canales. Este formato es uno de los formatos más comunes para almacenar audio digital (CD Audio), y nos proporcionará un espectro de frecuencias suficiente para el oído humano (de 0kHz a 20kHz) y un sonido estéreo gracias a los 2 canales.

### 3.2.1 Clases de TinySound

**TinySound** gracias a su sencilla librería nos permite exprimir toda su funcionalidad con solamente 1 clase y 2 interfaces, **TinySound**, **Music** y **Sound**.

#### TinySound

La clase **TinySound** es la clase principal y gracias a ella podemos cargar audios y trabajar con ellos. Para ello debemos inicializarla mediante su método ***init*** y ya podríamos trabajar con toda la funcionalidad que nos proporciona. En el momento en el que dejasemos de trabajar con TinySound debemos llamar al método ***shutdown***, que se encarga de limpiar la clase y los buffers que utiliza con el fin de que en próximos usos no existan problemas.
Para poder cargar audios **TinySound** trabaja con las funciones ***loadMusic*** y ***loadSound***, las cuales aceptan como argumentos, una URL, un fichero o el nombre de la ruta donde se ubica el fichero que queremos cargar.

Estas dos funciones pueden parecer similares ya que tienen el mismo objetivo, sin embargo, para comprender la diferencia entre ellas, explicaremos las interfaces **Music** y **Sound**.

#### Music

La interfaz **Music** está enfocada a audios que representan música como tal, y nos permite modificar las propiedades principales de estos, así como tratar a estos audios de la misma manera que haríamos en un reproductor.

Si un audio ha sido cargado mediante la interfaz **Music** podrá ser reproducido, pausado, parado y reanudado en cualquier momento gracias a las funciones ***play***, ***pause***, ***stop*** y ***resume*** respectivamente. Además de esto podremos modificar propiedades del audio como el volumen o el canal por donde recibimos el audio, ya sea el izquierdo o el derecho (Panoramización/Panning) gracias a las funciones ***setVolume*** y ***setPan***.

Esta interfaz cuenta con más funcionalidad, pero no entraremos en detalle ya que no hemos usado esta en nuestro proyecto.

#### Sound

La interfaz **Sound** a diferencia de **Music** está enfocada a SFX, es decir, efectos de sonido.

La principal diferencia entre estas interfaces es que **Sound** no permite que el audio que se ha cargado sea pausado, parado o resumido, ya que solamente podrá ser reproducido, y por lo tanto, una vez se llame a la función ***play*** este sonará hasta que termine. También cuenta

con una forma más sencilla de indicar el volumen o panoramización de un sonido, indicándose esto mediante los argumentos de la función ***play***.

## *3.3 Simulator*

**Simulator** es una herramienta basada en java y creada para este proyecto que nos permite crear ritmos de batería pseudoaleatorios mediante la utilización de una carpeta donde se almacenan todos los sonidos que pueden ser usados.

Gracias a **Simulator** somos capaces de crear simulaciones en las que podemos especificar distintos parámetros de los ritmos que van a ser generados. Estos parámetros pueden diferir dependiendo del fin que tengamos. Si queremos una composición "completamente" pseudoaleatoria solamente indicaremos el BPM (*Beats per minute*), el número de sonidos que compondrán la batería y la longitud del ritmo. Si queremos una simulación menos aleatoria, además podremos indicar, por cada tipo de *drum*, el número de elementos de ese tipo que estarán en el ritmo generado.

Para comprender esto de forma más visual, veremos dos ejemplos de simulación diferentes y explicaremos cada uno de sus argumentos.

Por ejemplo, si queremos una ejecución "completamente" pseudoaleatoria, estos serán los argumentos:

```
140.0 10 8
```

Los argumentos de esta ejecución son:

- **BPM**: 140.0
- **Numero de sonidos**: 10
- **Longitud del ritmo**: 8

El BPM hace referencia al número de beats/ritmos que hay por minuto y afecta principalmente a la velocidad a la que la canción se desarrolla. Para entenderlo de una forma muy sencilla, es exactamente lo mismo que las pulsaciones por minuto.

El número de sonidos hace referencia, como el propio nombre dice, al número de sonidos que se usarán en la composición.

Por último, la longitud del ritmo se especifica en barras/bars. Un bar está compuesto por 4 beats, y mediante el bpm y este parámetro podemos calcular la duración en segundos del ritmo generado.

$$\boldsymbol{Dur.beat\ (s)} = \left(\frac{\boldsymbol{BPM}}{\boldsymbol{60}}\right) ; \boldsymbol{Dur.N\ bars\ (s)} = \boldsymbol{N} * \boldsymbol{4} * \boldsymbol{Dur.beat}$$

Sin embargo, si queremos detallar más, tendremos unos argumentos como estos:

```
140.0 8 2 1 1 3 1 0 1
```

Los argumentos de esta ejecución son:

- **BPM**: 140.0
- **Longitud del ritmo:** 8
- **Número de Kicks:** 2
- **Número de Snares:** 1
- **Número de Claps:** 1
- **Número de HiHats Closed:** 3
- **Número de HiHats Open:** 1
- **Número de Cymbals:** 0
- **Número de Percusiones:** 1

Aquí los argumentos simplemente indican el número de elementos de cada tipo que compondrán el ritmo, por lo tanto, el parámetro número de sonidos desaparece.

A parte de estos métodos de ejecución, existen otros enfocados a las pruebas, los cuales serán mencionados posteriormente.

### 3.3.1 Clases de Simulator

El simulador cuenta con cuatro clases que se encargan de realizar toda la funcionalidad. Estas clases son:

- **DrumSound:** Parametriza los sonidos.
- **SimulationConfigurator:** Configura las simulaciones.
- **SimulationEvaluator:** Evalúa las simulaciones.
- **SoundRecorder:** Permite grabar audio interno, en este caso, las simulaciones.

Comenzamos hablando de la clase **DrumSound** ya que esta tiene un papel fundamental dentro de la clase principal, y posteriormente desarrollaremos la clase principal, es decir, **SimulationConfigurator.**

#### DrumSound

Como hemos mencionado antes, la clase **DrumSound** se encarga de almacenar todas aquellas características que definen a un sonido además del propio sonido en sí. Ciertas características son obtenidas dentro de la clase, y otras desde el constructor, por lo tanto, explicaremos el origen de todas estas mediante sus atributos y funciones.

##### Atributos de DrumSound

**DrumSound** se encarga de almacenar:

```
private String soundPath;
private Sound drumSound;
private int numReps;
private ArrayList<Integer> startTimes;
private int numPrincipalDivs;
private typeOfDrum drumType;
```

**Figura 3-1 –** Atributos clase DrumSound

- ***soundPath***: Ruta del sonido, es decir, donde se encuentra este en el sistema.
- ***drumSound***: El propio sonido en sí, que es almacenado en un objeto Sound de TinySound.
- ***numReps***: Número de repeticiones que puede tener un sonido como máximo en un beat (Como antes se ha mencionado 1 beat = ¼ bar)
- ***startTimes***: Instantes de tiempo donde el sonido será reproducido dentro de la simulación.
- ***numPrincipalDivs***: Número de veces en las que el sonido se encuentra en una división principal, es decir, en el instante exacto de tiempo en el que comienza cualquier beat de la simulación.
- ***drumType***: Tipo de sonido dentro de los elementos de una batería.

Antes de entrar en detalle con la procedencia de cada atributo, hablaremos del tipo **typeOfDrum**, ya que es el único que hasta ahora no conocemos.

**typeOfDrum** es una enumeración creada con el fin de poder identificar cada sonido existente con un elemento distinto de la batería que compone nuestra simulación. Estos elementos son aquellos que hemos visto en los argumentos de la ejecución detallada.

- ***Kicks***: Bombo
- ***Snares***: Tambores
- ***Claps***: Palmadas
- ***HiHatsClosed***: Charles/Contratiempos (Cerrados)
- ***HiHatsOpen***: Charles/Contratiempos (Abiertos)
- ***Percussion***: Percusiones (Toms y efectos varios)
- ***Cymbals***: Platillos Ride y Crash.

Gracias a esta enumeración podemos diferenciar cada sonido y por lo tanto, proporcionar distintas características a cada uno.

Con este tipo aclarado, pasamos a detallar la procedencia de cada atributo.

Los atributos ***soundPath, numReps, drumType*** **y** ***DrumSound*** proceden del constructor de la clase:

```
public DrumSound(String path, int numRepetitions, typeOfDrum type,
Sound sound) {
	this.soundPath = path;
	this.numReps = numRepetitions;
	this.startTimes = new ArrayList<Integer>();
	this.drumType = type;
	this.drumSound = sound;
	this.numPrincipalDivs = 0;
}
```

**Figura 3-2 –** Constructor de la clase DrumSound

Estos atributos han sido generados dentro de la clase simulationConfigurator, y cuando hablemos de ella en el siguiente punto explicaremos como se ha obtenido cada uno.

Como podemos observar en este, los otros atributos son inicializados, ya que hasta que no se haga uso de las funciones pertinentes no pueden tener ningún valor, por lo tanto, pasemos a explicar estas.

### Funciones de DrumSound

Las funciones de **DrumSound** están compuestas por dos métodos principales encargados de la obtención de los ***startTimes*** y de ***numReps***, y los métodos getter y setter pertinentes de los atributos que se necesitan obtener desde otras partes del código.

Las funciones principales son ***setDrumTimes*** y ***selectRandomBarMultiplyer***, y de forma conjunta actúan para obtener los instantes de tiempo en los que un sonido será reproducido gracias a, un objeto de java Random, que se encarga de seleccionar un multiplicador dentro de unos valores predefinidos, la duración de un beat en la simulación actual (en segundos) y el número de bars que definen a la simulación.

***setDrumTimes*** es la función pública, y, por lo tanto, aquella que se llama desde otras partes del código, mientras que ***selectRandomBarMultiplyer*** es privada y es invocada exclusivamente desde ***setDrumTimes.***

```
public void setDrumTimes(int durBeat, int numBars) {
	Random rand = new Random();
	int i = 0;

	if (this.drumType == typeOfDrum.HiHatsClosed || this.drumType ==
	typeOfDrum.HiHatsOpen) {
		// We set the times where the drum will be played.
		for (i=0; i < numBars; i++) {
			for (int k = 0; k < this.numReps; k++)
				this.startTimes.add(Math.round((durBeat*i*4) +
				(selectRandomBarMultiplyier(rand)*durBeat*4)));
		}
	}else {
		// We set the times where the drum will be played.
		for (i=0; i < numBars; i++)
			this.startTimes.add(Math.round((durBeat*i*4) +
			(selectRandomBarMultiplyier(rand)*durBeat*4)));
	}
}
```

**Figura 3-3 –** Función ocupada de añadir los tiempos a cada Drum

```
private float selectRandomBarMultiplyer(Random rand) {
	List<Float> sepValuesMain = Arrays.asList(0f, 0.125f, 0.25f, 0.375f,
	0.5f, 0.625f, 0.75f, 0.875f);

	List<Float> sepValuesReps = Arrays.asList(0f, 0.0625f, 0.125f,
	0.1875f, 0.25f, 0.3125f, 0.375f, 0.4375f, 0.5f, 0.5625f, 0.625f,
	0.6875f, 0.75f, 0.8125f, 0.875f, 0.9375f);

	float selectedValue;

	if (this.drumType == typeOfDrum.Kicks || this.drumType ==
	typeOfDrum.Snares || this.drumType == typeOfDrum.Claps ||
	this.drumType == typeOfDrum.Cymbals)
		selectedValue = sepValuesMain.get(rand.nextInt(sepValues-
		Main.size()));

	else
		selectedValue = sepValuesReps.get(rand.nextInt(sepValues-
		Reps.size()));

	if(selectedValue == 0f || selectedValue == 0.25f || selectedValue ==
	0.5f || selectedValue == 0.75f) {
		this.numPrincipalDivs += 1;
	}

	return selectedValue;
}
```

**Figura 3-4** – Funcion que selecciona aleatoriamente el beat

Como podemos observar en el código de ***setDrumTimes*** se comprueba si el sonido es de tipo ***HiHatsClosed*** o ***HiHatsOpen***, esto es debido a que estos elementos son los únicos que tienen un número de repeticiones por bar superior a uno. Esta decisión se tomó debido a que, en las composiciones de batería, por norma general, estos son los elementos que presentan más repetición. Otros elementos podrían disponer de esta propiedad también, sin embargo, el control sería más complejo, ya que habría bars en los que, si debería ocurrir y otros en los que no, y, al ser una composición con tiempos aleatorios, podría generar resultados malos debido a excesos de repetición en la mayor parte de las simulaciones.

Una vez se ha comprobado el tipo de sonido, elegiremos N instantes de tiempo donde será reproducido gracias al bucle for del que la función dispone. Estos instantes de tiempo estarán cada uno dentro de un bar distinto, y por ello el bucle depende del número de bars de la simulación. Para elegir el instante donde sonará, tenemos que tener en cuenta que las composiciones que estamos realizando tienen un compás 4/4, lo que significa que nuestros beats están divididos en 4 partes llamadas steps, y nuestros bars estarán compuestos de 4 beats, por lo tanto, tendrán 16 steps. Esto significa, que, si sabemos la duración de un bar en segundos, simplemente tendremos que multiplicar esta por un “multiplicador” y así obtendremos el instante de tiempo exacto en el que el sonido se situará y reproducirá dentro de un bar.

Los multiplicadores son obtenidos gracias a ***selectRandomBarMultiplyer,*** que contiene dos arrays desde donde selecciona estos de forma aleatoria gracias al objeto Random de java que mencionamos anteriormente. Disponemos de dos arrays ya que, cada elemento

tiene sus posiciones asignadas por norma general a unos lugares concretos dentro de un bar. Por ello, se hace distinción entre, las posiciones para los ***Kicks, Snares, Claps y Cymbals***, contenidas en *sepValuesMain,* y las posiciones para ***HiHatsClosed, HiHatsOpen, y Percusiones,*** contenidas en *sepValuesReps*.

Los valores que contienen estos arrays son los multiplicadores pertinentes para dividir un bar en 16 steps. Como hemos mencionado antes, estos multiplicadores corresponden a un compás 4/4, sin embargo, en el caso de querer cambiar esto a otro tipo de compás, como por ejemplo 3/4, simplemente tendríamos que dividir cada bar en 4 beats con 3 steps cada uno, lo cual daría un total de 12 multiplicadores. Estos serían los nuevos valores para este compás:

```
List<Float> sepValues = Arrays.asList(0f, 0.0833f, 0.0166f, 0.25f, 0.333f,
0.4166f, 0.5f. 0.5833f, 0.666f, 0.75f, 0.833f, 0.9166f);
```

**Figura 3-5 –** Valores correspondientes a un compás 3/4

Además de la selección de multiplicadores, esta función se encarga de comprobar si los tiempos donde se reproducirá un sonido corresponde a una división principal. Esto también tiene que ver con los compases, ya que, dependiendo del compás, las divisiones principales son unas u otras.

Consideramos división principal, aquel instante en el que comienza cada beat de un bar, es decir, si nuestro compás tiene 4 beats por bar, tendremos cuatro divisiones principales, indicadas por los multiplicadores: 0, 0.25, 0.5 y 0.75.

En el caso de tener un compás 6/8, tendríamos 8 beats por bar, y por lo tanto, 8 divisiones principales: 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75 y 0.875.
De esta manera, cada vez que un multiplicador seleccionado corresponde a una división principal, aumentamos el número de divisiones principales del sonido en cuestión. Este atributo nos será útil más adelante en la evaluación de la simulación.

Una vez hemos seleccionado el multiplicador, lo devolvemos a la función ***setDrumTimes*** y pasamos a calcular el instante de tiempo mediante la siguiente fórmula:

$$\boldsymbol{Instante\ t.(s) = (Dur.beat * i * 4) + (mult * Dur.beat * 4)}$$

La letra i representa la iteración del bucle en la que nos encontramos, y, sabemos que esta depende del número de bars, por lo tanto, gracias a esta primera parte de la fórmula podemos controlar en todo momento el bar en el que nos encontremos, ya que siempre nos devolverá el instante inicial de este. Con la segunda parte de la fórmula obtenemos el instante de tiempo dónde se reproducirá el sonido dentro del bar que le corresponda.

Por ejemplo, si estamos en la iteración/bar 2, nuestro beat dura 10 segundos y obtenemos el multiplicador 0.125, el instante de tiempo donde se reproducira nuestro sonido será:

$$\boldsymbol{Instante\ t.(s) = (10 * 2 * 4) + (0.125 * 10 * 4) = 80 + 5 = Segundo\ 85}$$

Una vez el tiempo de reproducción se ha calculado, este se añade a la variable ***startTimes***, y, tras todas las iteraciones del bucle, ya dispondrá de todos los instantes y estará lista para ser usada en la simulación.

### SimulationConfigurator

Esta clase es nuestra clase principal, ya que, en ella se almacena y evalúa todo aquello que ocurre durante la simulación. Para poder llevar a cabo esta funcionalidad es necesario organizar esta clase con unos atributos concretos y unas funciones que nos proporcionen de forma adecuada aquello que necesitamos, por lo tanto, describiremos en detalle los elementos que componen esta clase con el fin de mostrar el flujo que sigue esta clase a la hora de crear una simulación.

#### Atributos de SimulationConfigurator

SimulationConfigurator se encarga de almacenar:

```
private float bpm;
private int numberOfSounds;
private int numberOfBars;
private ArrayList<DrumSound> sounds;
private SortedMap<Integer, List<DrumSound>> soundsOrdered;
private SimulationEvaluation evaluation;
```

**Figura 3-6 –** Atributos de la clase SimulationConfigurator

- ***bpm***: Beats por minuto de la simulación.
- ***numberOfSounds***: Número de sonidos de la simulación.
- ***numberOfBars***: Número de bars de la simulación.
- ***sounds***: Sonidos que serán reproducidos en la simulación.
- ***soundsOrdered***: Sonidos que serán reproducidos en la simulación ordenados por su tiempo de reproducción.
- ***evaluation*:** Evaluación que contiene el número de sonidos de cada tipo y la propia evaluación en sí.

Al igual que pasa con la clase **DrumSound** no todos los atributos están establecidos desde que se crea el objeto, por lo tanto, explicaremos la procedencia de todos estos y la funcionalidad que hay detrás de ellos.

Los atributos ***bpm, numberOfSounds y numberOfBars*** se obtienen gracias al constructor de la clase:

```
public SimulationConfigurator(float beatsPerMinute, int numSounds, int
numBars) {
	this.bpm = beatsPerMinute;
	this.numberOfSounds = numSounds;
	this.numberOfBars = numBars;
	this.sounds = new ArrayList<DrumSound>();
	this.soundsOrdered = new TreeMap<Integer, List<DrumSound>>();
	this.evaluation = new SimulationEvaluation();
}
```

**Figura 3-7 –** Constructor de la clase SimulationConfigurator

Como podemos observar, estos se encuentran en los argumentos del constructor y provienen del programa principal (Main), donde han sido obtenido mediante los argumentos de la ejecución.

El atributo ***sounds*** se inicializa como una lista de **DrumSound**, ya que aquí guardaremos todos los sonidos que se vayan a utilizar durante la reproducción, sin embargo, no serán llamados desde aquí, porque sería necesario realizar ordenaciones de los tiempos mientras nuestra simulación se está reproduciendo, y, por ende, podría acarrear problemas de retardo en la reproducción. Para solucionar este problema, existe el atributo ***soundsOrdered***, donde gracias a la funcionalidad existente en la clase podremos almacenar todos los sonidos de la simulación de forma ordenada respecto a su tiempo de ejecución, eliminando de esta manera los problemas que se nos presentaban anteriormente. Por último, ***evaluation*** se inicializa a un objeto del tipo evaluación, del que hablaremos más adelante.

### Funciones de SimulationConfigurator

**SimulationConfigurator** cuenta con funciones que podemos dividir en dos tipos, aquellas que sirven para configurar la evaluación, es decir, parametrizar esta y obtener todos los sonidos que la componen, y, por otra parte, aquellas que evaluan a la simulación. Primero nos centraremos en las funciones que consiguen que ***soundsOrdered*** contenga todos los elementos necesarios para la reproducción, y posteriormente comentaremos las demás.

Comenzamos con la función ***soundRandomizer,*** la cual se encarga de obtener de forma pseudo-aleatoria los sonidos que compondrán nuestra simulación gracias a un objeto Random de java. Posee tres argumentos, un objeto BufferedWriter para escribir resultados en un fichero, un Array de Strings que dependiendo del método de ejecución del simulador puede contener, o bien, los argumentos del segundo tipo de ejecución, o una cadena vacía, y, por último, un Boolean que solamente valdrá true sí la llamada a la función ha sido realizada con el fin de optimizar la simulación. Hablaremos de este parámetro cuando expliquemos nuestro método de optimización.

Existen dos métodos de ejecución para esta función:

1) Indicamos número de elementos de cada tipo, y sonidos aleatorios.
2) Número de elementos de cada tipo y sonidos aleatorios.

Para obtener los sonidos de forma aleatoria en ambos métodos, utilizamos un bucle while con la siguiente condición:

```
while(this.sounds.size() < this.numberOfSounds)
```

Esto significa que, mientras nuestro atributo ***sounds*** no contenga el número de sonidos que se le ha asignado a la simulación, se realizará lo siguiente:

1) Seleccionamos el tipo de sonido, para ello utilizamos el enumerado **typeOfDrum** del que hablamos anteriormente ya que dispone de una función estática que selecciona aleatoriamente uno de sus elementos gracias al objeto Random que se le pasa por argumento. (***Este paso no se realiza para el primer método)***

```
public static typeOfDrum getRandomDrum(Random random) {
      return values()[random.nextInt(values().length)];
}
```

**Figura 3-8 –** Función de selección aleatoria de Drum

2) Una vez tenemos el tipo de sonido, creamos la ruta al objeto y abrimos esta en un fichero File. La ruta no nos lleva directamente a un sonido concreto, si no que abre en el fichero File el directorio que contiene todos los elementos del tipo seleccionado. Aquí es donde volvemos a utilizar nuestro objeto Random para que seleccione uno de los ficheros existentes en la carpeta como sonido para la simulación.

```
// Execution mode 1
sound = "/Drums/<tipoElemento>";

// Execution mode 2
sound = "/Drums/" + typeOfDrum.getRandomDrum(rand).name();

File dir = new File("Sounds" + sound);
File[] files = dir.listFiles();
File file = files[rand.nextInt(files.length)];
```

**Figura 3-9 –** Construcción de la ruta del Drum elegido

3) Con la ruta del sonido, comprobamos el tipo de este con el fin de establecer un numero de repeticiones para el sonido, y el **typeOfDrum** correspondiente, que serán guardados en unas variables auxiliares (***En método 1 no hace falta la comprobación***).

   Como comentamos antes en el apartado de **DrumSound**, los únicos elementos que contienen un número de repeticiones mayor a 1 son los ***HiHatsOpen*** y los ***HiHatsClosed***.

   El número de repeticiones para los ***HiHatsClosed*** será entre 1 y 4 y el de los ***HiHatsOpen*** entre 1 y 2. Este número se obtiene de nuevo gracias a nuestro objeto Random, mediante la función ***nextInt***.

4) Creamos un objeto **DrumSound** con las variables auxiliares, la ruta del sonido y el sonido cargado mediante el módulo **TinySound** (*Usamos TinySound.LoadSound)*, establecemos sus ***startTimes*** mediante una llamada al método ***setDrumTimes*** y añadimos este a nuestro atributo ***sounds***.

```
auxDrum = new DrumSound(auxPath, numReps, auxType, TinySound.loadSound(auxPath));
auxDrum.setDrumTimes(this.getDurationBeat(), this.numberOfBars);
this.sounds.add(auxDrum);
```

**Figura 3-10 –** Construcción del Drum y establecimiento de tiempos aleatorios

Con el bucle finalizado, ya tenemos nuestra variable ***sounds*** completa, y, por lo tanto, ya podemos ordenarla para su posterior reproducción. Esta ordenación se realiza mediante la función ***orderTimes.*** Después de la llamada a este método, la función ***soundRandomizer*** termina, y, al ser de tipo void no devuelve nada.

***orderTimes*** es la función encargada de ordenar los sonidos por sus tiempos de reproducción y de almacenarlos en el atributo **soundsOrdered**. Posee un único argumento del tipo Boolean, este es el mismo argumento que el tercer argumento de la función ***soundRandomizer***, y su uso está relacionado con la optimización del problema.

***soundsOrdered*** es del tipo *Map<Integer, List<DrumSound>*, y almacena pares clave-valor, dónde la clave es un instante de tiempo, y el valor es un objeto de tipo *List<DrumSound>* que representa los sonidos que se deben reproducir en ese instante.

Esto lo realiza gracias a una variable interna auxiliar de tipo *List<DrumSound>* donde almacenamos todos nuestros sonidos y a un bucle anidado con la siguiente estructura:

```
for (int i = 0; i < this.numberOfBars; i++) {
      for(DrumSound drum: this.sounds) {
            //Código interno.
      }
}
```

**Figura 3-11 –** Bucle de ordenamiento de tiempos

Este bucle actúa ordenando los sonidos bar a bar, de una forma similar a como lo hacíamos en ***setDrumTimes***, por ello recorremos en un bucle interno todos los sonidos que tenemos.

Como usamos un objeto Map, debemos de tener cuidado a la hora de ir añadiendo objetos, ya que, si una clave no existe e intentamos añadir un valor, obtendremos una excepción. Por lo tanto, cada vez que vamos a añadir un sonido comprobamos si el instante de tiempo que le corresponde a este ya existe dentro del objeto Map. Si existe, añadimos este al valor (*Nuestra Lista de sonidos, List<DrumSound>*), y si no, creamos una nueva entrada en el Map con este instante como clave y una lista nueva con el sonido dentro como valor.

```
//If the key exist, we add a value.
if (this.soundsOrdered.containsKey(drum.getStartTimes().get(i))) {

      this.soundsOrdered.get(drum.getStartTimes().get(i)).add(drum);
}
//If not, we create a new key, and the value with it.
else {
      auxList = new ArrayList<DrumSound>();
      auxList.add(drum);
      this.soundsOrdered.put(drum.getStartTimes().get(i), auxList);
}
```

**Figura 3-12 –** Rellenado del objeto de tipo Map *soundsOrdered*

Sin embargo, tenemos que controlar otro detalle más, ya que, como sabemos los sonidos de tipo ***HiHatsClosed/Open*** pueden tener un número de repeticiones superior a 1. La solución a esto es muy sencilla, simplemente comprobamos el tipo de sonido al comienzo del bucle interno, si no es del tipo ***HiHatsClosed/Open*** realizamos lo mostrado en la anterior figura, si lo es, debemos realizar esto, pero de una forma distinta.

Al tener repeticiones, hay que realizar estas comprobaciones, pero para cada repetición, por lo que creamos un tercer bucle que dependerá del número de repeticiones del sonido. Sin tener en cuenta este nuevo bucle, la única diferencia respecto a los sonidos que no tienen repetición es la forma en la que añadimos estos al atributo. Mientras que para el resto de elementos añadimos directamente el tiempo y el sonido una vez por bar, aquí

podemos añadirlos varias veces por bar, por lo que hay que tener en cuenta el número de repeticiones y el bar en el que nos encontramos.

Esto lo conseguimos controlar de la siguiente forma:

```
this.soundsOrdered.get(drum.getStartTimes().get((numRepsAux*i)+k)).add(drum);
```

**Figura 3-13 –** Añadido de drums del tipo HiHatsClosed/Open

Aquí, *i* representa el bar en el que nos encontramos, *k* la repetición que estamos añadiendo y *numRepsAux* el número de repeticiones que existen en total. De esta manera, conseguimos indexar correctamente todos los tiempos de los ***HiHatsClosed/Open,*** y, por lo tanto, no hay problemas a la hora de añadirlos.

Con los tiempos ya ordenados, el atributo ***soundsOrdered*** ya está listo, y podremos reproducir nuestra simulación de forma sencilla y ordenada. Esta reproducción se lleva a cabo mediante la función ***playSimulation***.

Esta función se encarga de reproducir la simulación con los sonidos que se encuentran en el atributo ***soundsOrdered***. Esta reproducción es muy sencilla, ya que, gracias al trabajo que hemos realizado antes ordenando los sonidos por su tiempo de reproducción, solamente debemos controlar el tiempo entre sonidos. Este control lo realizamos creando una variable auxiliar que guarda el tiempo en el que se reprodujo el último sonido, por lo que, la inicializamos a 0 (*oldTime*).

Con esta variable creada comenzamos a recorrer una a una las entradas del Map de ***soundsOrdered***. Para cada entrada, realizamos una espera mediante la siguiente llamada:

**Thread.sleep(entry.getKey() – oldTime)**

Una vez se ha realizado la espera, guardamos como nuevo *oldTime* el valor de *entry*.***getKey,*** y reproducimos todos los sonidos que correspondan a la clave de la entrada del mapa.

De esta manera, conseguimos respetar el tiempo entre los instantes de reproducción y ser fieles a los tiempos que se habían establecido.

Con estas funciones explicadas, damos por finalizada la clase ***SimulationConfigurator***, ya que, el resto de funciones que posee son métodos getter/setter que no necesitan ser explicados.

### SimulationEvaluation

Uno de los problemas que presenta evaluar nuestras simulaciones es que quizás el punto más importante a la hora de evaluar estas es la opinión del usuario, algo que, un ordenador no puede realizar automáticamente sin pedirlo por pantalla. Esto además de implicar que no podríamos usar un algoritmo genético de forma que optimizásemos nuestra función de evaluación automáticamente, implicaría tener que evaluar composición a composición a mano, algo que, sin duda, es muy costoso si se quiere contar con un conjunto de datos grande.

Por ello, se ha desarrollado un método automático que evalúa las características generales que una batería debería cumplir para las composiciones más frecuentes. Este método se llama ***automatedEvaluation,*** y se realiza para cada simulación, obteniendo una nota entre 0 y 10. Utiliza un único argumento, la propia simulación.

La evaluación se divide en tres evaluaciones independientes que se ocupan de evaluar aspectos concretos de la simulación. Cada una de estas evaluaciones tiene su propia puntuación, y una vez se obtienen las tres se suman y se obtiene el total.

**1)** La evaluación general se evalúa sobre 2.5 puntos, y se encarga de comprobar los aspectos básicos de la batería a partir de los elementos que la componen. Se utiliza el número de elementos de cada tipo para comprobar que exista variedad y a la vez equilibrio. Esta comprobación se realiza mediante condicionales que responden a las siguientes preguntas:

- *¿Existe al menos un Kick?*
- *¿Existe al menos un Snare o un Clap?*
- *¿Existe al menos un HiHatClosed?*
- *¿Existe al menos un HiHatOpen?*
- *¿Existen como máximo 2 sonidos de cada tipo?*

Si se responde con un sí, se suma un punto, mientras que, si se responde con un no, se resta 0.125 puntos, excepto en la última pregunta donde se resta 0.5.

Con la puntuación obtenida, multiplicamos esta por 2.5 y la dividimos por 5 para obtener la nota que le corresponde sobre 2.5 puntos.

De esta primera evaluación se encarga la función ***generalEvaluation***.

**2)** La evaluación detallada se evalúa sobre 6 puntos, y se encarga de comprobar la sobrecarga existente en cada instante en el que se reproduce uno o varios sonidos, la sobrecarga que existe en cada bar de la simulación y el silencio existente dentro de la simulación.

Consideramos que existe sobrecarga de sonidos en un instante concreto si existen más de 3 sonidos, que hay sobrecarga en nuestro bar si dentro de este se reproducen 7 o más sonidos y que hay silencio si durante 2.5 beats no se ha reproducido ningún sonido.

Cada vez que una de estas condiciones se cumple, aumentamos un punto a unas variables que se encargan del conteo de estas características. Además de esto, por cada 4 veces que ocurra esto añadimos otro punto a la variable correspondiente.

Una vez tenemos las variables con el número de fallos, aplicamos la siguiente fórmula para obtener tres notas distintas sobre 10, calculamos la media aritmética de estas y obtenemos la nota final sobre 6.

Para la sobrecarga en un instante de tiempo:

$$\boldsymbol{Nota} = \left(\left(\mathbf{1} - \left(\frac{\boldsymbol{numSobrecargaInst}}{\boldsymbol{tamaño\ soundsOrdered}}\right)\right) * \mathbf{10}\right)$$

Para el silencio y la sobrecarga: (Ejemplo con silencio)

$$\boldsymbol{Nota = \left(1 - \left(\frac{numSilencio}{numBars}\right)\right) * 10}$$

Sumamos las 3 notas, multiplicamos por 6 y dividimos por 30, obteniendo así nuestra puntuación detallada.

De esta evaluación se encarga el método ***detailedEvaluation***. Utiliza como único argumento la simulación, ya que necesita información sobre los sonidos que la componen, así como la duración de un beat o el número de bars de esta.

**3)** La evaluación por divisiones principales se evalúa sobre 1.5 puntos y se encarga de comprobar si los elementos principales de nuestra simulación (***kicks, snares/claps***) se encuentran dentro de las divisiones principales de cada bar. Esta evaluación es muy importante ya que si estos elementos no se encuentran en estas posiciones no marcarán el ritmo de la simulación de la forma correcta, sin embargo, los tiempos son decididos de forma aleatoria, por lo que, es de esperar que las puntuaciones aquí no siempre sean altas. Consideramos este método como aquel que diferencia los ritmos buenos de los perfectos.

Este método evalúa de forma muy sencilla gracias a que cuando establecimos los tiempos en cada **DrumSound,** guardábamos el número de divisiones principales de cada elemento.

Para explicar este método, utilizaremos el siguiente árbol de decisión, y explicaremos las distintas evaluaciones:

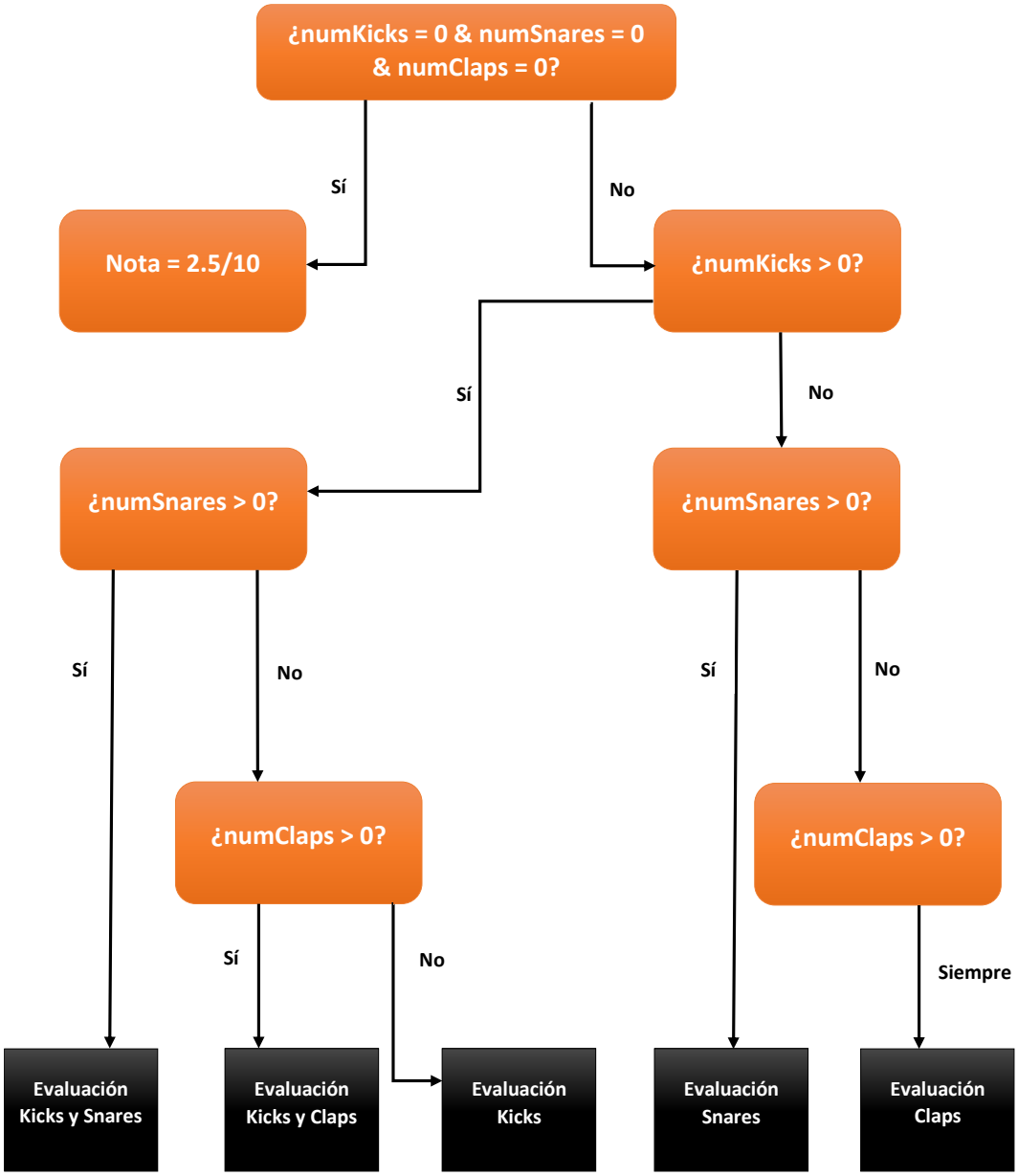



**Figura 3-14** – Árbol de decisión para la evaluación general

Las evaluaciones que encontramos en las hojas de árbol, calculan la suma del número de divisiones principales de los elementos que están en estas dentro de nuestra simulación. Por ejemplo, si acabamos en la hoja, *Evaluación Kicks y Claps*, sumaremos el número de divisiones principales de los ***kicks*** y ***claps*** que componen nuestra simulación.

Con el número de divisiones principales total pasamos a calcular la nota mediante la siguiente fórmula:

$$\boldsymbol{Nota\ divisiones} = (\frac{\boldsymbol{Num.\ de\ divisiones\ principales} * \boldsymbol{12.5}}{\boldsymbol{Num.\ de\ bars} * \boldsymbol{4}})$$

A diferencia de otras fórmulas para calcular la nota, aquí multiplicamos el numerador por 12.5, esto se debe a que la probabilidad de obtener una buena nota en esta evaluación es muy baja, y aquello que se evalúa depende enteramente de la aleatoriedad de la simulación, por lo que se incrementa la nota multiplicando por un valor más alto al máximo. En caso de superar la nota máxima, la nota máxima será 10.

Esta nota final será multiplicada por 1.5 y dividida por 10, obteniendo así nuestra nota sobre 1.5 puntos.

La función encargada de esto es ***divisionsEvaluation***, y, al igual que ***detailedEvaluation***, tiene sólo un argumento, la propia simulación, que se encarga de crear un array de **DrumSound** que contiene los ***kicks*** de la simulación a evaluar, otro que contiene los ***snares*** y un último que contiene los ***claps***. Estos argumentos son los que se evalúan dentro del árbol de decisión.

### SoundRecorder

La clase ***SoundRecorder*** está basada en la Java Sound API y nos permite grabar un input de audio que provenga de los dispositivos de grabación del sistema en un fichero de audio de tipo WAV con el formato que el usuario desee.

Toda la información referente a esta clase se ha conseguido gracias a un artículo de la web codeJava [**18**], y podemos encontrar el código de esta en el anexo **B.**

## *3.4 Método de optimización*

A la hora de determinar el método que usamos para maximizar nuestro problema, debemos tener en cuenta la estructura de este, en nuestro caso, los atributos que lo forman y aquellos parámetros que juegan un papel importante en el resultado final. Dependiendo del formato de estos podremos usar un método u otro.

Nuestro problema depende completamente de parámetros aleatorios que son generados durante la creación del propio problema en sí. Al tratarse de algo tan difuso, no podemos saber con certeza que parámetros pueden causar un mayor impacto en las evaluaciones, por lo tanto, es necesario utilizar un método que pueda utilizar estos parámetros a su antojo con el fin de optimizar al máximo cada simulación generada.

Por todo esto, debemos utilizar un método que sea capaz de resolver problemas que no dispongan de un *algoritmo o heurística que nos proporcione la solución satisfactoria*. Estos métodos son los conocidos como metaheurísticas, y nos permiten *resolver un tipo de problema de computacional general, usando los parámetros dados por el usuario sobre unos procedimientos genéricos y abstractos de una manera que se espera eficiente.* (Metaheurística, s.f)

La metaheurística elegida para este caso es el Algoritmo de **recocido simulado**. Esta elección ha sido llevada a cabo gracias a la sencillez y eficiencia que proporciona en la mayoría de problemas de optimización. Uno de los ejemplos más conocidos es el problema del viajante [**19**]. Este algoritmo está basado en el "*proceso de recocido del acero y*

*cerámicas, una técnica que consiste en calentar y luego enfriar lentamente el material para variar sus propiedades físicas*" (Algoritmo de recocido simulado, s.f.)

La variación de las propiedades físicas representa el cambio de estados, que, en el caso del acero, se produce debido a que "*el calor causa que los átomos aumenten su energía y que puedan así desplazarse de sus posiciones iniciales*" (Algoritmo de recocido simulado, s.f). Este desplazamiento representa el cambio a un nuevo estado en el que alguna propiedad ha variado. Por lo tanto, si aplicamos este conocimiento a nuestro problema, podríamos considerarlo un cambio en alguno de los atributos o parámetros que componen nuestro problema.

### 3.4.1 Implementación de la metaheurística elegida

Gracias a un artículo publicado sobre este método de optimización [**20**], y, al pseudocódigo encontrado en el Anexo **A**, la implementación de la metaheurística ha sido muy sencilla, ya que solamente hemos tenido que adaptar nuestro problema al ejemplo propuesto en el artículo.

Para ello, hemos creado dentro de nuestro programa principal (Main), un tercer modo de ejecución. Este modo de ejecución a diferencia de los anteriores funciona sin argumentos.

Al no tener argumentos, todas las elecciones a la hora de crear la simulación, excepto la de la duración de esta, que siempre será 8, son aleatorias. Esto quiere decir que, Beats per minute, número de sonidos, y los propios sonidos de la simulación han tenido que ser aleatorizados. Actualmente, sabemos cómo se obtienen los sonidos de forma aleatoria en una simulación gracias a la función ***soundRandomizer***, por lo que solamente explicaremos la aleatoriedad de las variables *bpm* y *numSounds*.

Estas variables son obtenidas de la siguiente forma:

```
List<Float> selectBPM = Arrays.asList(90f, 95f, 100f, 105f, 110f,
115f, 120f, 125f, 130f, 135f, 140f, 145f, 150f, 155f, 160f, 165f,
170f, 175f, 180f, 185f, 190f, 195f, 200f);

// Initial parameters for SA
final float bpm = selectBPM.get(rand.nextInt(selectBPM.size()));
final int numSounds = rand.nextInt(((20-3)+1)-3);
```

**Figura 3-15 –** Selección aleatorio de parámetros para la simulación

*Bpm* obtiene su valor, al igual que *numSounds* gracias a un objeto de tipo Random, sin embargo, cada uno lo obtiene de forma distinta. En el caso de *bpm*, se elige un valor aleatorio del array *selectBPM*. Los valores contenidos en este array son genéricos, y se podría introducir cualquier valor que se desease probar. *numSounds*, por otro parte, obtiene un número aleatorio entre 4 y 20.

Una vez hemos elegido estos parámetros, declaramos las variables que necesitamos para realizar el algoritmo. Estas variables son:

- *Temp:* Representa la temperatura inicial a la que comienza el algoritmo
- *CoolingRate:* Ratio de descenso de temperatura por iteración.
- *CurrentSolution:* Almacena la simulación con la que se trabaja en cada iteración.
- *Best:* Almacena la mejor simulación encontrada.

*currentSolution* y *best* representan estados y sus propiedades serán alteradas varias veces mientras se realiza el algoritmo. Al comenzar el algoritmo, estas variables contienen el mismo valor, la simulación inicial, que será creada con los parámetros aleatorios y obtendrá sus sonidos gracias al método ***soundRandomizer***.

El algoritmo sigue el siguiente diagrama:

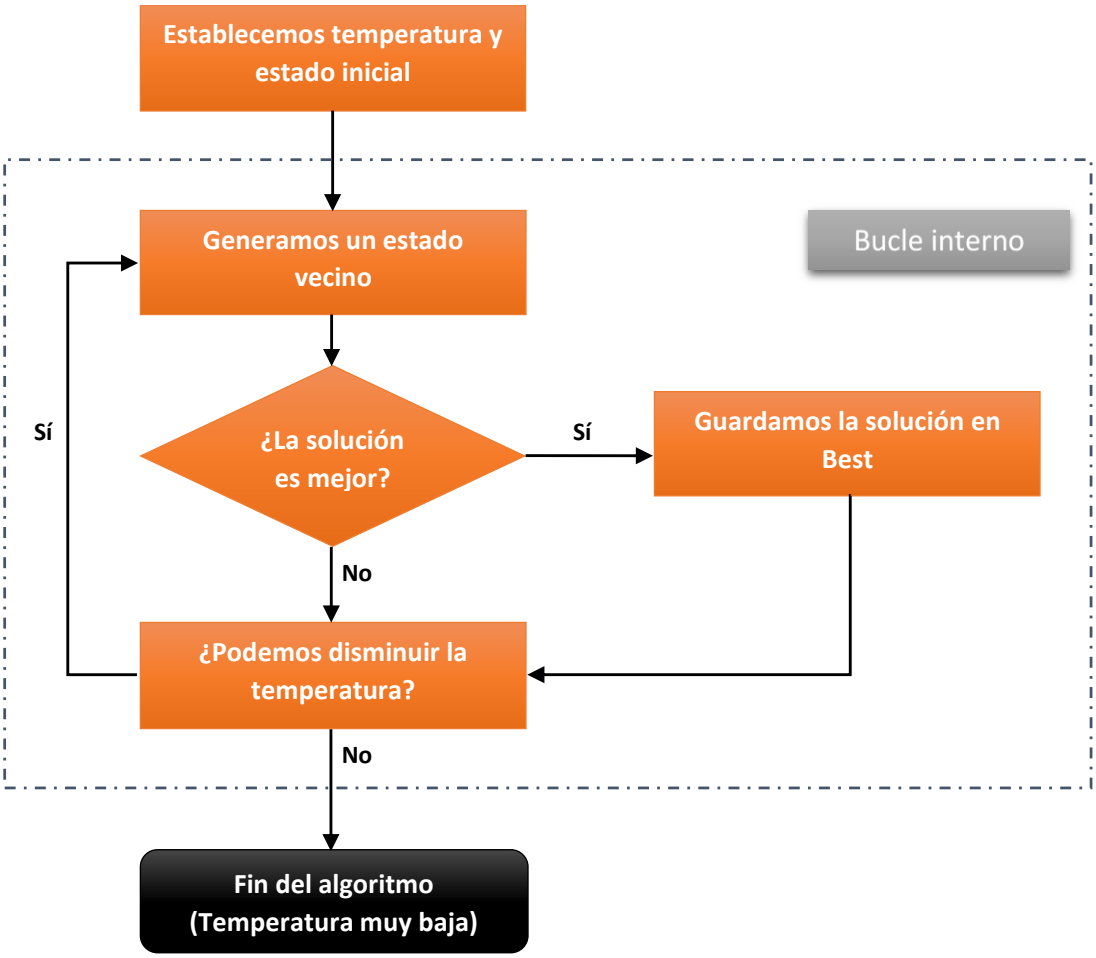



**Figura 3-16 –** Algoritmo de recocido simulado mediante un diagrama de estados

El diagrama muestra de forma general el funcionamiento del algoritmo. Para nuestro problema, debemos diferenciar ciertos aspectos. Los estados representan simulaciones, y los vecinos, simulaciones que han recibido alguna alteración en sus atributos. También, es importante conocer los valores de la temperatura inicial y, la ratio de enfriamiento. La temperatura inicial (*temp)* comienza con un valor de 10000, mientras que, la ratio (*coolingRate)* comienza con 0.003.

El bucle del algoritmo sigue la condición:

```
while (temp > 1) {
      // Código interno
}
```

**Figura 3-17 –** Condición del bucle del algoritmo de recocido simulado

Dentro del bucle creamos estados vecinos alterando la simulación guardada en *currentSimulation* y guardándola en otra variable a la que llamaremos *newSolution*. Las propiedades que han sido alteradas dependen de la iteración en la que nos encontremos, ya que, de cada 5 iteraciones que realiza el bucle, las primeras 4 ejecutan una alteración que llamaremos **ALT-1**, y la quinta, realiza otra alteración distinta, que llamaremos **ALT-2**.

**ALT-1** realiza una alteración sobre los sonidos que componen la simulación. Se lleva a cabo mediante la eliminación de uno de los sonidos gracias a un índice aleatorio que actúa sobre el atributo **soundsOrdered**, y la adición de un nuevo sonido que, de nuevo, se ha generado de forma aleatoria.

**ALT-2** actúa de forma similar a **ALT-1**, sin embargo, la alteración que se realiza es sobre el atributo **numSounds**. Se elige un nuevo número de sonidos para la simulación, lo que implica la generación de nuevos sonidos mediante ***soundRandomizer***. Este método tiene un impacto mucho mayor que **ALT-1** debido al brusco cambio que realiza sobre la simulación, por ello se realiza cada 5 iteraciones.

Una vez hemos realiza la alteración correspondiente, calculamos la energía de cada simulación. La energía la obtenemos mediante la siguiente fórmula:

$$\boldsymbol{Energía\ (Simulación) = (10 - Evaluacion(Simulacion))}$$

Esto es así, ya que lo que se pretende es encontrar la mínima energía posible. Con las energías obtenidas, comparamos mediante una función que representa la probabilidad de transición al nuevo estado vecino. La función compara las energías de la siguiente forma:

```
if (newEnergy < energy) {
    return 1.0;
}

double aux = Math.exp((energy - newEnergy) / temperature);

return aux;
```

**Figura 3-18 –** Codificación de la probabilidad de aceptación del algoritmo

*newEnergy* representa la energía de la simulación vecina, y *energy* la de la simulación actual.

Si la nueva energía es superior a la actual, la probabilidad de transición es 100%, mientras que, si no lo es, la probabilidad depende de la fórmula mostrada en la figura **3.16.**

Con la transición resuelta, actualizamos los valores de *currentSimulation*, y de *best* en el caso de que la energía de *currentSimulation* sea menor que la de *best*.

Una vez actualizados todos estos valores solamente queda disminuir la temperatura. La disminución se realiza mediante la siguiente fórmula:

$$\boldsymbol{Temperatura = Temperatura * (1 - ratio\ de\ enfriamiento)}$$

Una vez el bucle ha terminado, dentro de la variable *best* se encontrará la simulación con la menor energía, lo que significa que habremos obtenido una simulación optimizada y con una evaluación superior a la que obtendríamos sin realizar este algoritmo.

# 4 Experimentos y resultados

A lo largo de este capítulo veremos los experimentos realizados y los resultados que se han obtenido en estos con el fin de analizar el funcionamiento de nuestro proyecto en distintas situaciones. La primera situación a analizar será la generación pseudoaleatoria de simulaciones sin optimización, y, la segunda, la generación de una simulación óptima mediante nuestro método de optimización.

## *4.1 Generación de simulaciones pseudoaleatorias*

Para comprobar el funcionamiento de este tipo de generaciones, dentro del programa Main se crea un nuevo modo de ejecución que cuenta con dos argumentos. El número de simulaciones a generar, y cada cuantas iteraciones se aumenta el BPM en 5.

Estos son los resultados para una prueba pequeña de 20 ejecuciones y aumento de BPM cada iteración.

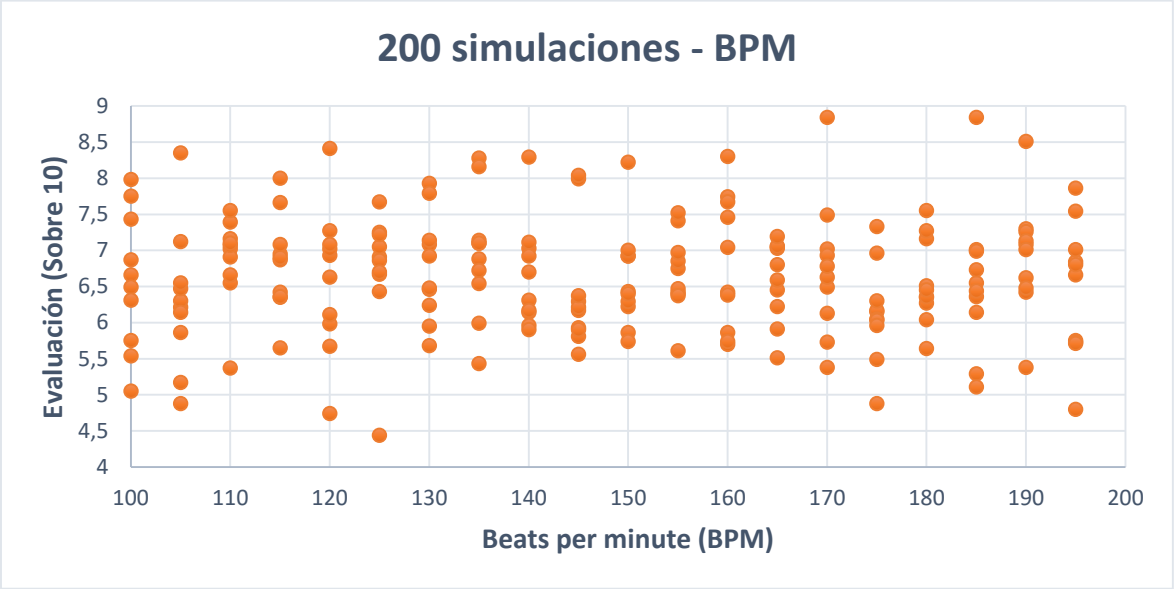
| BPM | Número de sonidos | Kicks | Snares | Claps | HiHats Closed | HiHats Open | Cymbals | Percussion | Evaluación |
|---|---|---|---|---|---|---|---|---|---|
| 100 | 11 | 5 | 0 | 3 | 1 | 0 | 2 | 0 | 7.11 |
| **105** | **5** | **2** | **2** | **0** | **1** | **0** | **0** | **0** | **8.86** |
| 110 | 10 | 3 | 0 | 1 | 2 | 2 | 1 | 1 | 7.58 |
| 115 | 10 | 2 | 1 | 1 | 1 | 1 | 0 | 4 | 6.89 |
| 120 | 14 | 3 | 2 | 4 | 2 | 0 | 1 | 2 | 6.25 |
| 125 | 19 | 0 | 1 | 4 | 5 | 0 | 6 | 3 | 4.38 |
| 130 | 12 | 0 | 1 | 2 | 2 | 2 | 1 | 4 | 5.37 |
| 135 | 9 | 2 | 2 | 2 | 1 | 0 | 0 | 2 | 7.01 |
| 140 | 3 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 4.92 |
| 145 | 20 | 7 | 3 | 5 | 0 | 0 | 2 | 3 | 5.75 |
| 150 | 17 | 1 | 3 | 2 | 4 | 3 | 1 | 3 | 6.61 |
| 155 | 7 | 0 | 1 | 0 | 0 | 3 | 2 | 1 | 6.42 |
| 160 | 10 | 1 | 3 | 2 | 1 | 2 | 1 | 0 | 6.84 |
| 165 | 12 | 3 | 3 | 0 | 0 | 4 | 0 | 2 | 7.43 |
| 170 | 19 | 5 | 2 | 1 | 5 | 3 | 2 | 1 | 7.07 |
| 175 | 11 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 6.18 |
| 180 | 8 | 0 | 2 | 1 | 0 | 1 | 1 | 3 | 5.64 |
| 185 | 17 | 6 | 2 | 3 | 0 | 2 | 3 | 1 | 7.02 |
| 190 | 17 | 1 | 4 | 4 | 3 | 1 | 1 | 3 | 7.02 |
| 195 | 19 | 0 | 1 | 4 | 3 | 4 | 2 | 5 | 5.22 |

**Tabla 4.1** – Resultados de 20 ejecuciones diferentes

Como podemos observar, todos los parámetros, a excepción del BPM son aleatorios, lo cual nos indica que las generaciones se están realizando correctamente. Las evaluaciones, como ya sabemos, dependen, a parte de estos parámetros, de otros que están relacionados con los **startTimes** de cada sonido.

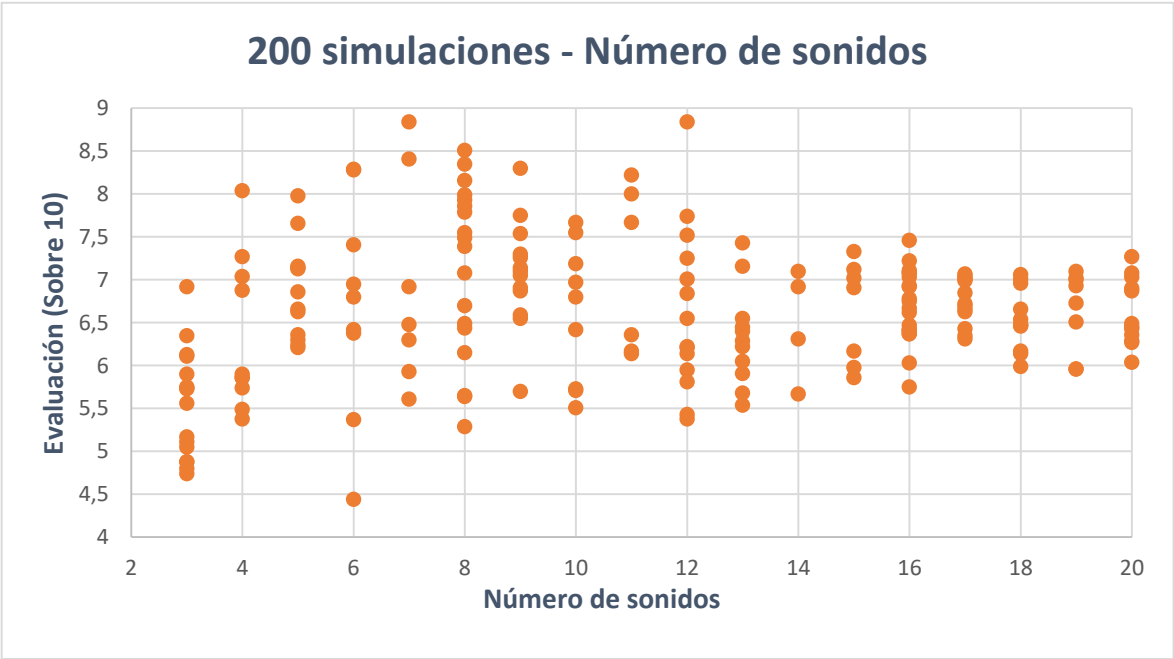
## *4.2 Evaluación de simulaciones pseudoaleatorias*

Nuestra función de evaluación actúa según varios parámetros, por lo que vamos a graficar estos frente a la evaluación de las simulaciones generadas, para observar la importancia de cada uno.




**Gráfica 4-1** – Representación evaluación frente al *BPM*

Los beats per minute o BPM, como podemos observar, a pesar de que su valor varíe, obtiene evaluaciones de todo tipo. En ningún momento se muestra un BPM en el que se encuentren resultados por encima del resto. Esto tiene una explicación muy sencilla, este parámetro no afecta en nada a la función de evaluación, solamente representa el tempo de las simulaciones, por lo que, es lógico que se obtengan para todos los valores, evaluaciones tanto buenas, como malas.

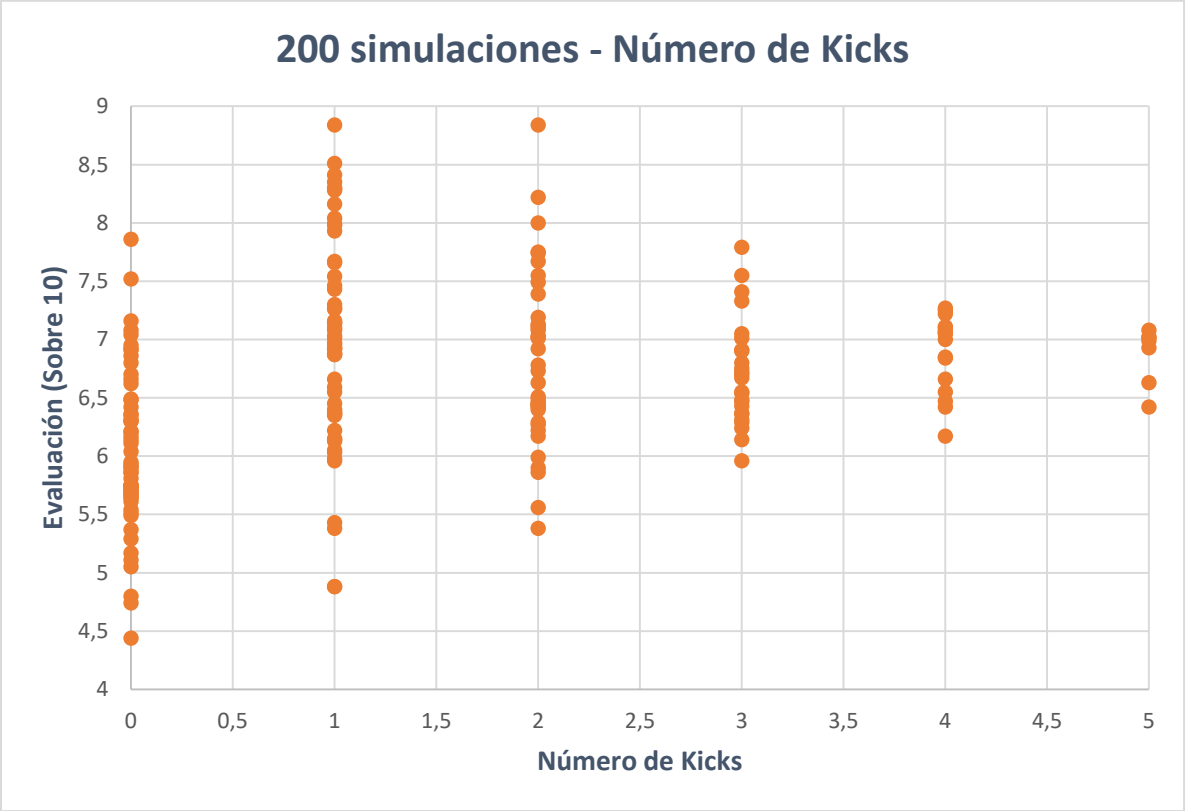



**Gráfica 4-2** – Representación evaluación frente al *Número de sonidos*

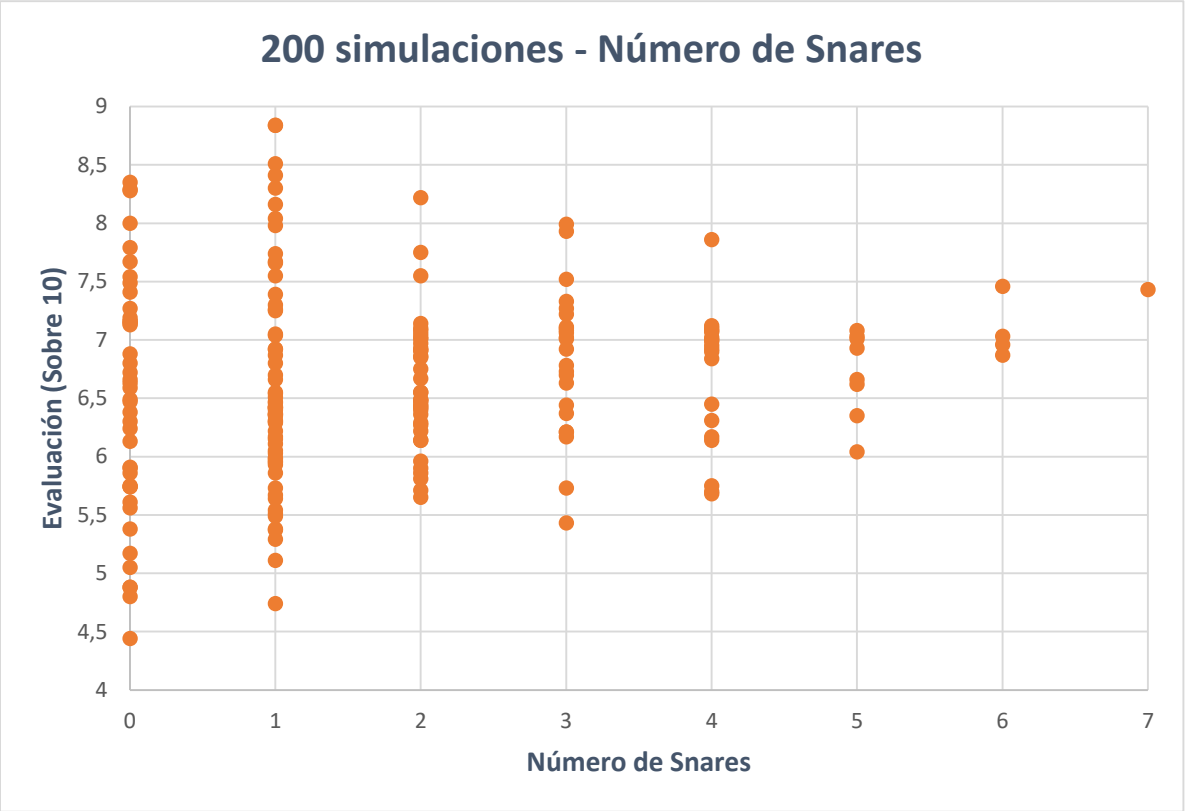
Con el número de sonidos podemos ver más dependencia en las evaluaciones. Las peores evaluaciones se obtienen con el mínimo número de sonidos, en este caso 3, que coincide con el mínimo número posible establecido en las simulaciones. Desde este punto hasta los 13 sonidos podemos observar un comportamiento similar en todas las simulaciones, obteniendo todas estas unas evaluaciones en un rango de valores similar. A partir de los 13 de sonidos, parece representar el mismo comportamiento para todas las simulaciones.

Esto resultados tienen sentido, ya que es un atributo que representa una parte importante de la función de evaluación. Cuando existen pocos sonidos, la evaluación general y la evaluación de divisiones tienen menos posibilidades de obtener una puntuación alta, mientras que a más sonidos haya estas pueden aumentar, mientras que la detallada puede disminuir un poco. Por estas razones las evaluaciones con un número de sonidos medio son aquellas que obtienen las mejores puntuaciones, mientras que los extremos siempre tendrán un límite.

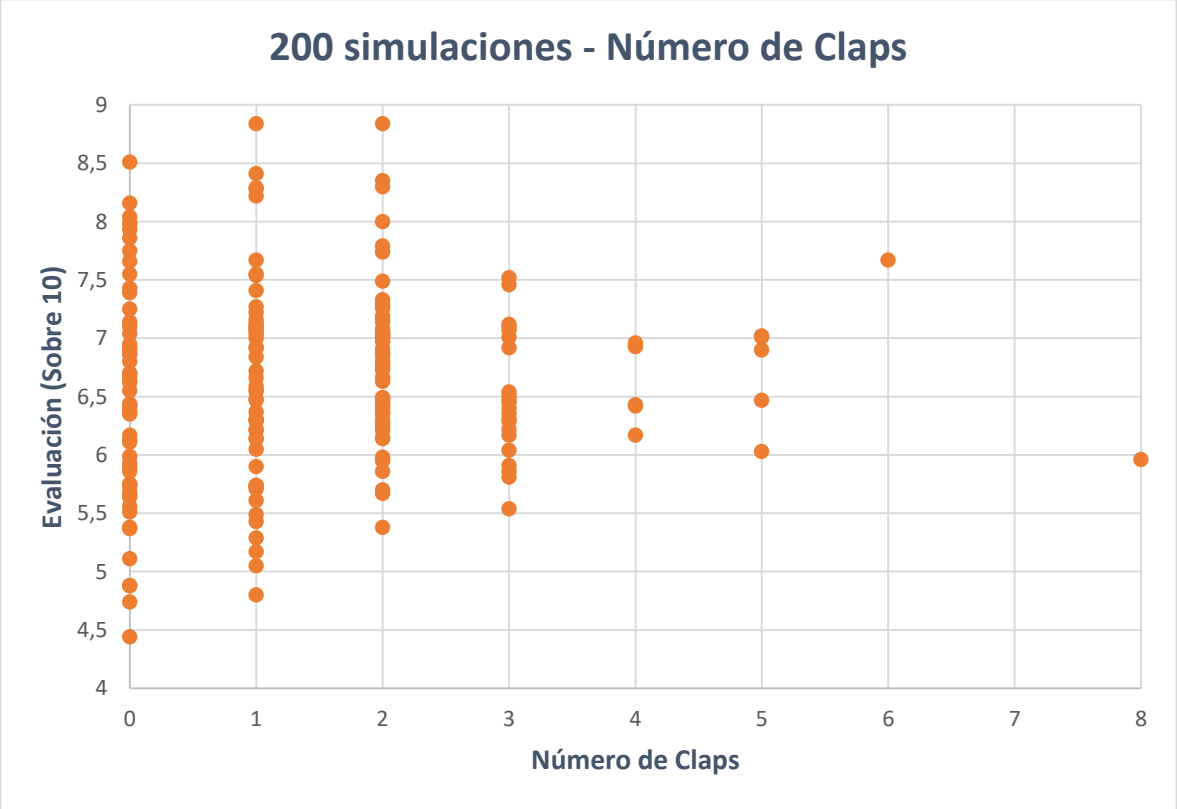
Esta explicación nos sirve en general para todos los atributos que representan el número de veces que aparece un tipo de sonido en concreto. Sin embargo, es importante ver las diferencias que pueden existir entre ellos, por lo que se mostrarán también las gráficas relativas a estos a continuación.



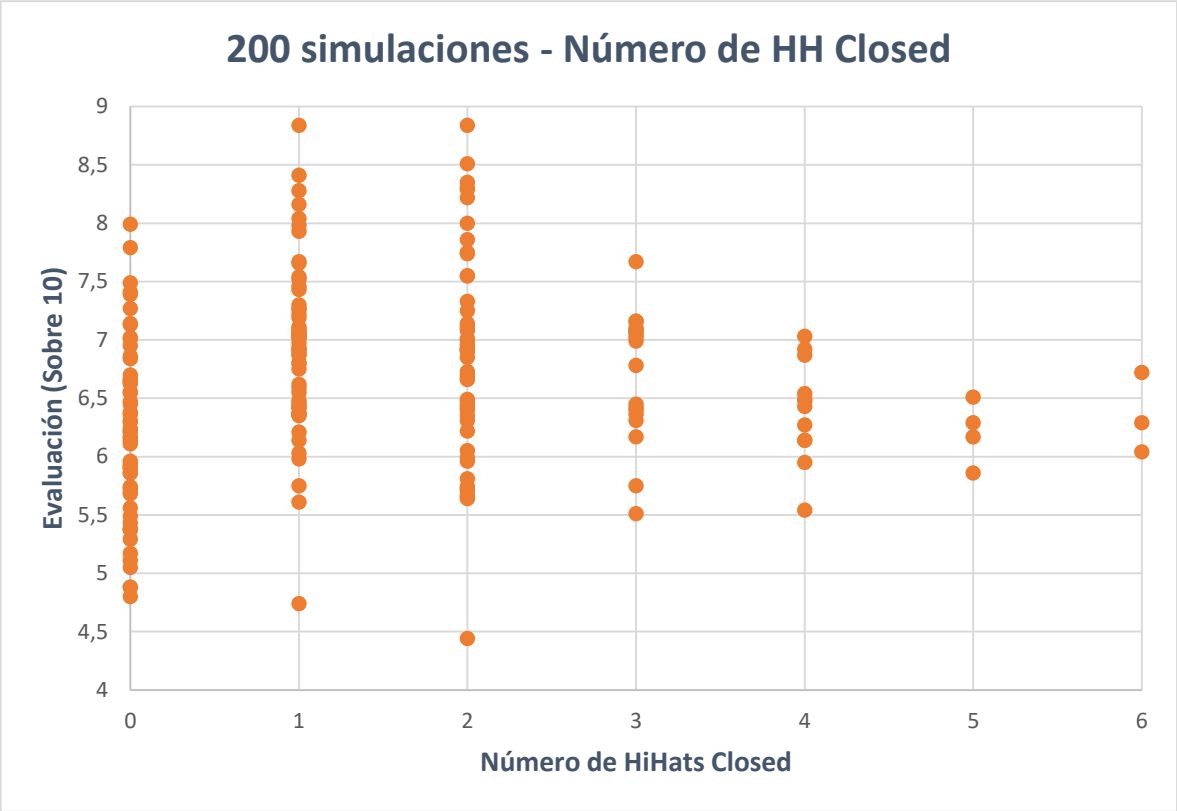

**Gráfica 4-3 –** Representación evaluación frente al atributo *Kicks*



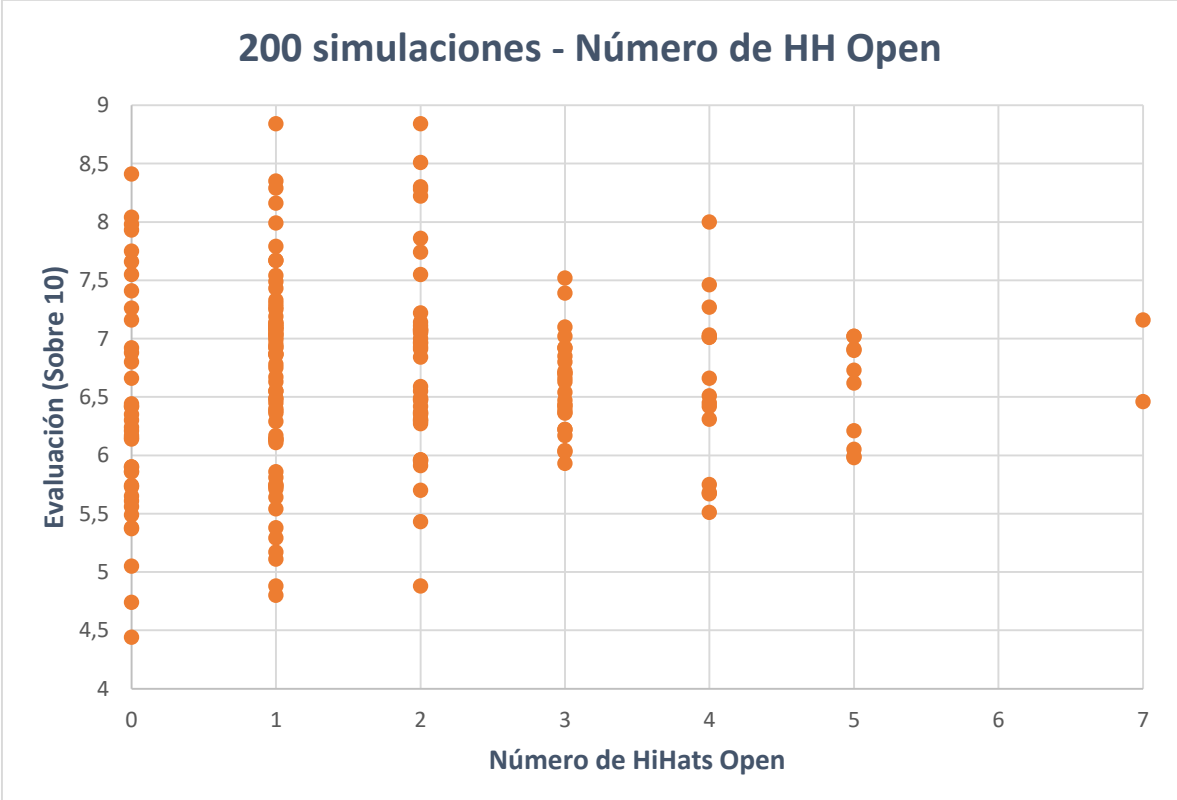

**Gráfica 4-5 –** Representación evaluación frente al atributo *Snares*



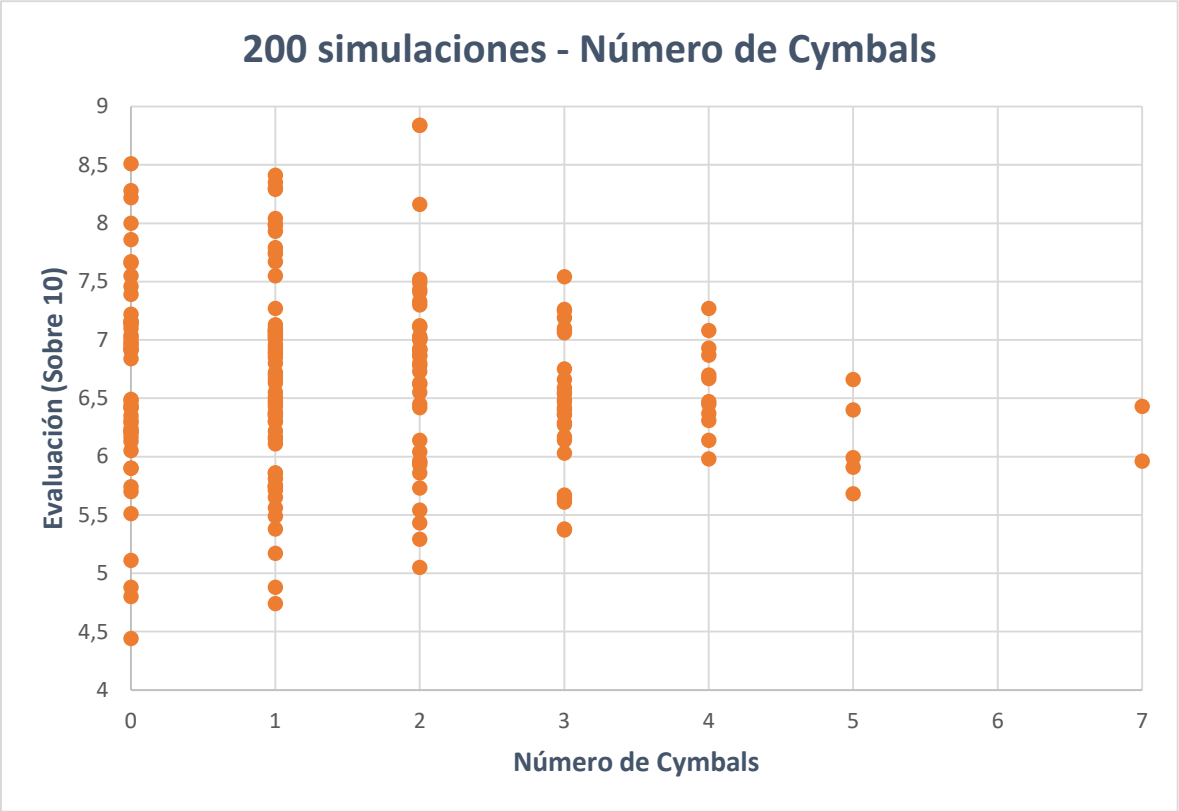

**Gráfica 4-4 –** Representación evaluación frente al atributo *Claps*



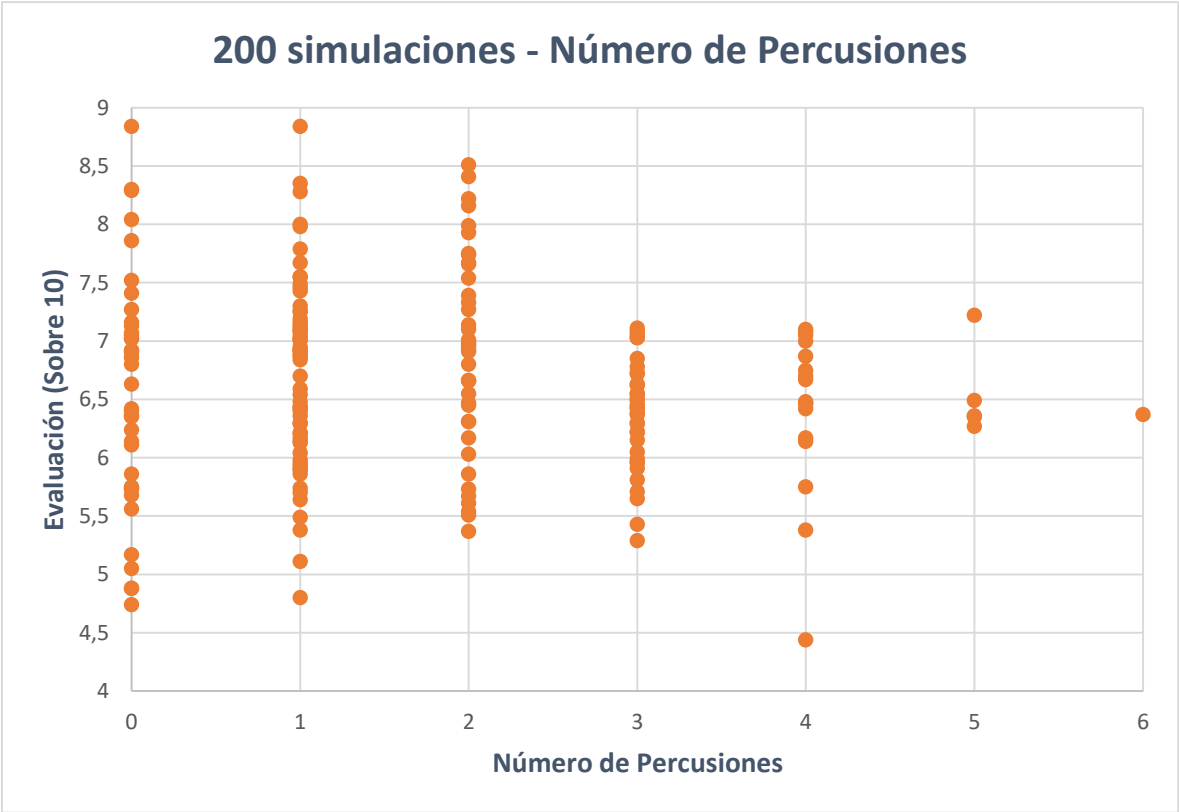

**Gráfica 4-7 –** Representación evaluación frente al atributo *HiHats Closed*




**Gráfica 4-6 –** Representación evaluación frente al atributo *HiHats Open*




**Gráfica 4-8 –** Representación evaluación frente al atributo *Cymbals*




**Gráfica 4-9 –** Representación evaluación frente al atributo *Percusiones*

Claramente podemos observar como las mejores evaluaciones en todos los casos corresponden a un número de elementos bajo, siendo siempre los mejores valores entre 0 y 2.

Esto no significa que las evaluaciones necesiten tener pocos sonidos para ser buenas, pero suelen obtener mejores resultados cuando el número de elementos de cada tipo no es muy grande.

Como mencionamos en la sección 3.2, en el método de evaluación, la evaluación general puntúa sobre 2.5 y depende enteramente del número de sonidos total y de cada tipo. Cuando no existen más de dos sonidos de cada tipo, obtenemos las mejores puntuaciones, y en el resto de casos obtenemos puntuaciones más bajas. Esto podemos observarlo en todas las gráficas a partir del valor 3 en el eje X.

*Para ver más en detalle estos valores, los resultados de la prueba grande con 200 iteraciones y cambio de BPM cada 10 iteraciones se encuentra en el anexo **D**.*

## *4.3 Generación de simulaciones óptimas*

Para la obtención de una simulación óptima hacemos uso del modo de ejecución descrito en la sección **3.3**. Mostramos los resultados de 10 simulaciones óptimas:

| BPM | Número de sonidos | Kicks | Snares | Claps | HiHats Closed | HiHats Open | Cymbals | Percussion | Evaluación |
|---|---|---|---|---|---|---|---|---|---|
| 90 | 9 | 3 | 1 | 3 | 2 | 1 | 0 | 0 | 9.25 |
| 95 | 8 | 2 | 2 | 2 | 1 | 2 | 2 | 0 | 9.11 |
| 125 | 8 | 2 | 0 | 1 | 1 | 1 | 2 | 1 | 9.17 |
| 135 | 7 | 2 | 0 | 1 | 1 | 2 | 1 | 0 | 9.25 |
| 150 | 9 | 2 | 2 | 2 | 1 | 2 | 2 | 1 | 9.1 |
| 155 | 9 | 2 | 1 | 2 | 2 | 2 | 2 | 1 | 9.32 |
| **160** | **8** | **2** | **0** | **2** | **1** | **2** | **2** | **1** | **9.66** |
| 165 | 7 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 9.21 |
| 170 | 7 | 2 | 2 | 2 | 1 | 1 | 0 | 1 | 9.42 |
| 170 | 8 | 2 | 2 | 0 | 1 | 1 | 1 | 1 | 9.36 |
| 180 | 7 | 1 | 2 | 2 | 2 | 2 | 2 | 0 | 9.03 |

**Tabla 4.2 –** Resultados de 10 ejecuciones óptimas

Podemos observar unos resultados mucho mayores a los que se obtienen en las generaciones sin optimización. El mínimo resultado encontrado en la prueba del anexo **D** es 4.44 puntos, y el máximo obtenido en esta tabla es 9.66. A pesar de no que estas puntuaciones no provengan de la misma ejecución nos muestran la cantidad de mejora que puede llegar a proporcionarnos nuestro método de optimización, siendo la mejora:

$$\text{Mejora} = \frac{\text{Puntuación máxima}}{\text{Puntuación mínima}} = \frac{9.66}{4.44} = 2.176$$

Podemos considerar que, de forma general, este método puede optimizar y proporcionar en las simulaciones hasta 2 veces la puntuación obtenida inicialmente, lo que supone una mejora considerable.

Si nos fijamos en los valores de los atributos, el análisis de los resultados en la sección 4.1 coincide con lo obtenido en la tabla. Como hemos mencionado antes, el BPM es independiente del resultado, el número de sonidos se mantiene entre 7 y 9, valores pertenecientes al mejor rango posible, y el resto de atributos, excepto en la primera simulación, se mantienen con un valor de 2 o menos.

# 5 Conclusiones y trabajo futuro

Con el desarrollo del simulador terminado, y los resultados ya analizados, podemos obtener ciertas conclusiones sobre este. Para estas conclusiones nos referiremos a los objetivos, asunciones e hipótesis que hicimos en la sección **3.1**.

## *5.1 Conclusiones*

Los objetivos se han cumplido satisfactoriamente. Nuestro simulador genera simulaciones pseudoaleatorias mediante una biblioteca de sonidos, y, permite la elección de parámetros según la ejecución que se quiera realizar. El simulador dispone de una función de evaluación automatizable que realiza en función de sus atributos y parámetros sin necesidad de que el usuario proporcione ningún tipo de información. Por último, gracias al algoritmo de recocido simulado, hemos sido capaces de maximizar la función de evaluación definida, y, de esta manera, obtener mejores resultados.

Las asunciones realizadas pueden considerarse satisfactorias a pesar de que una de ellas no se ha cumplido, la función de evaluación no tiene siempre forma suave. Los cambios pequeños pueden llegar a suponer un cambio grande en la función de evaluación ya que esta evalúa según los sonidos de los que dispone, y, si, por ejemplo, cambiamos un elemento principal por uno menos importante, digamos, un Kick por un Cymbal, la evaluación podría llegar a disminuir mucho debido a la evaluación general y de divisiones principales.

El resto de asunciones como ya hemos dicho, se cumplen. Generamos distintas simulaciones cada ejecución, la evaluación es muy barata, algo que hemos podido observar con la rapidez con la que hemos obtenido los resultados de 2000 simulaciones, y, por último, gracias a las gráficas marginales de la sección **4.2**, podemos confirmar que existe ruido en la evaluación.

Respecto a las hipótesis, concretamos que, por norma general, las evaluaciones con una nota alta corresponden a composiciones complejas y bien definidas, a pesar de existir casos en los que evaluaciones altas no siempre corresponden a composiciones agradables de escuchar, y, sin embargo, evaluaciones bajas sí. La metaheurística si ha resultado ser un buen método de optimización, ya que, en la sección **4.3** observamos cómo podemos llegar a mejorar nuestra nota hasta más del doble de su valor.

Finalmente, como esperábamos, las restricciones se han cumplido, y, esperamos poder desarrollar esa funcionalidad en un trabajo futuro.

## *5.2 Trabajo futuro*

Como ya hemos mencionado en las conclusiones, las posibilidades para añadir funcionalidad son muchas, sin embargo, aquellas que serían primordiales para continuar con el desarrollo del proyecto antes de entrar en detalles serías las siguientes:

- **Implementación de nuevos instrumentos:** La inclusión de nuevos instrumentos a parte de la batería como pianos, bajos, violines… Podría crearse una clase que simulase un objeto MIDI [**21**] a la que se le asignase una onda de sonido con el fin de simular cualquier sonido posible, además de poder componer melodías que posteriormente podrían ser exportadas a otros softwares de edición/producción de audio.

- **Soporte para la creación y modificación de ondas de sonido:** Gracias a esto, nuestro simulador podría disponer de un sintetizador [**22**], lo que significa que podríamos generar nuestros propios "*instrumentos*", ya fuera de forma aleatoria o no. Además de esto podríamos tratar cualquier sonido que utilice el simulador y cambiar sus características a nuestro antojo.

- **Implementación de selección de compás para las simulaciones:** Como hablamos en la sección **3.2.1**, nuestras simulaciones funcionan con un compás 4/4, por lo que, la implementación de nuevos compases proporcionaría más variedad a las generaciones.

- **Biblioteca de sonidos más compleja:** Incluir un sistema de etiquetado por las características de cada sonido que le permitiese al simulador realizar selecciones más correctas. Junto a este etiquetado podría añadirse una separación en los sonidos por géneros musicales. Estas diferencias harían al simulador menos aleatorio en lo que a selección se refiere, sin embargo, podrían generarse simulaciones enfocadas a lo que el usuario quiere. No es una funcionalidad que sustituya a la actual, sino que, sería añadida.

- **Sistema de evaluación más complejo:** Sin duda, uno de los puntos de más indecisión a lo largo del desarrollo ha sido el sistema de evaluación, ya que actúa según unas reglas muy generales, y, a pesar de proporcionar buenos resultados en la mayoría de los casos, se podría realizar una evaluación mucho más compleja. Este nuevo sistema podría evaluar cosas como la armonía entre sonidos que se reproducen a la vez o, evaluar cada bar de la simulación y obtener una nota media de todos.

  Además de mejorar el existente, tendría que crearse un sistema de evaluación para la parte melódica de la simulación, es decir, aquella que implementa los instrumentos o sintetizadores de los que hemos hablado en puntos anteriores.

- **Método de optimización más complejo:** Nuestro método de optimización proporciona resultados muy buenos, pero, al igual que con el sistema de evaluación, seguramente se podría hacer uso de una técnica con más variaciones y formas de "mutación" en las simulaciones, por ejemplo, se podría hacer un barajado con los bars de una simulación y se obtendría una simulación diferente, que posteriormente sería comparada con la simulación actual.

Estas nuevas implementaciones no necesariamente deberían ser un método de optimización matemática, también podrían tratarse de un algoritmo genético.

# Referencias


[1] Música electrónica (Sin fecha). En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Música_electrónica

[2] García, A. (Sin fecha). Técnicas metaheurísticas. Recuperado de http://www.iol.etsii.upm.es/arch/metaheuristicas.pdf

[3] Algoritmo de recocido simulado. (Sin fecha). En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Algoritmo_de_recocido_simulado

[4] Melián, B. (2006, 11 de Septiembre). Introducción a la búsqueda tabú. Recuperado de http://leeds-faculty.colorado.edu/glover/fred%20pubs/329%20-%20Introduccion%20a%20la%20Busqueda%20Tabu%20TS_Spanish%20w%20Belen%2811-9-06%29.pdf

[5] Agudo, S. (2017, 25 de Enero). Qué se puede conseguir a día de hoy con un algoritmo componiendo música. *Genbeta*. Recuperado de https://www.genbeta.com/a-fondo/que-se-puede-conseguir-a-dia-de-hoy-con-un-algoritmo-componiendo-musica

[6] Musikalisches Würsfelspiel (Sin fecha). Recuperado de http://www.amaranthpublishing.com/MozartDiceGame.htm

[7] Proceso estocástico (Sin fecha). En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Proceso_estocástico

[8] McCormack, J. (1996). Grammar Based Music Composition. Recuperado de http://users.monash.edu/~jonmc/research/Papers/L-systemsMusic.pdf

[9] Algoritmos geneticos. (Sin fecha). Recuperado de http://eddyalfaro.galeon.com/geneticos.html

[10] Brikkemper, F. (2016, 5 de Octubre). Analyzing six deep learning tools for music generation. *The asimov institute*. Recuperado de http://www.asimovinstitute.org/analyzing-deep-learning-tools-music/

[11] Mantione, P. (2018, 16 de Abril). Orb Composer by Hexachords: Composing with AI [Review]. *Pro audio files*. Recuperado de https://theproaudiofiles.com/hexachords-orb-composer-review/

[12] Estación de trabajo de audio digital. (Sin fecha). En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Estación_de_trabajo_de_audio_digital

[13] Amper Music: 'In the year 2117, AI-generated music will be old hat'. (2017, 11 de Agosto). Recuperado de http://musically.com/2017/08/11/amper-ai-generated-music/

[14] Gestal, M. (2013, Agosto). Introducción a los Algoritmos Geneticos. Recuperado de https://www.researchgate.net/publication/237812449_Introduccion_a_los_Algoritmos_Geneticos

[15] Goldberg, D. (1989). Genetic Algorithms in search, Optimization & Machine Learning. Recuperado de http://www2.fiit.stuba.sk/~kvasnicka/Free%20books/Goldberg_Genetic_Algorithms_in_Search.pdf

[16] Sancho, F. (2016, 17 de Noviembre). Algoritmos de hormigas y el problema del viajante. *Fernando Sancho Caparrini*. Recuperado de http://www.cs.us.es/~fsancho/?e=71

[17] Diagrama de Gantt. (Sin fecha). En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Diagrama_de_Gantt.

[18] Capture and record sound into WAV file with Java Sound API (2017, 20 de Noviembre). Recuperado de http://www.codejava.net/coding/capture-and-record-sound-into-wav-file-with-java-sound-api

[19] Problema del viajante (Sin fecha). En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Problema_del_viajante

[20] Jacobson, L. (2013, 11 de Abril). Simulated Annealing for begineers. *The project spot.* Recuperado de http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6

[21] MIDI. (Sin fecha). En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/MIDI

[22] Sintetizador. (Sin fecha). En Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Sintetizador

# Glosario

| | |
|---|---|
| Algoritmo | Conjunto ordenado de operaciones sistemáticas que permite hacer un cálculo y hallar la solución de un tipo de problemas. |
| Panoramización | Técnica para enviar una señal de sonido en un medio estéreo o multicanal. |
| Proceso estocástico | Concepto matemático que sirve para usar magnitudes aleatorias que varían con el tiempo o para caracterizar una sucesión de variables aleatorias que evolucionan en función de otra variable, generalmente el tiempo. |
| Pseudocódigo | Lenguaje simplificado entre el programador y la máquina, hecho por el programador en su propio idioma, para describir un algoritmo y poder comprender mejor la estructura de dicho programa. |
| Simulación | Proceso de diseñar un modelo de un sistema real y llevar a término experiencias con él, con la finalidad de comprender el comportamiento del sistema. |
| Tempo (BPM) | Velocidad con la que debe ejecutarse una pieza musical. |

# Anexos

## A Pseudocódigo algoritmos de optimización

Este pseudocódigo [**3**] muestra la evolución del algoritmo de recocido simulado.

**1)** $Sea\ S\ =\ S0$
**2)** $Para\ k\ =\ 0$ hasta $k_{max}$
   1. $T\ =\ temperatura(k/k_{max})$
   2. $S_{nuevo} = Vecino(S)$
   3. $Si\ P(\ E(S), E(S_{nuevo}), T) \geq azar(0,1),\ entonces$:
      i. $S\ =\ S_{nuevo}$
**3)** $Devolver\ S$

Variables:

- S: Representa estados, siendo S0 el inicial.
- k: Número de iteraciones, siendo kmax el número de iteraciones máximas.
- T: Temperatura actual

Funciones:

- Vecino(S): Se obtiene un vecino aleatorio de S.
- P(E(S), E(Snuevo), T): Probabilidad de aceptación.

La probabilidad de aceptación es:

$$\boldsymbol{Si\ e' < e, entonces, P(e, e', T) = 1}$$
$$\boldsymbol{Si\ e' > e, entonces, P(e, e', T) = \exp(-\frac{e' - e}{T})}$$

Los siguientes pseudocódigos vienen en menor detalle respecto al recocido simulado ya que estos no se usan durante el proyecto y sólo son mencionados durante el estado del arte.

Pseudocódigo de algoritmos genéticos, obtenido de [**14**]:

```
Inicializar población actual aleatoriamente

MIENTRAS no se cumpla el criterio de terminación

        Crear población temporal vacía

   MIENTRAS población temporal no llena

         Seleccionar padres
         Cruzar padres con probabilidad Pc

         SI se ha producido el cruce

              Mutar uno de los descendientes con probabilidad Pm
              Evaluar descendientes
              Añadir descendientes a la población temporal

         SINO

              Añadir padres a la población temporal

         FIN SI
   FIN MIENTRAS

   Aumentar contador generaciones
   Establecer como nueva población actual la población temporal

FIN MIENTRAS
```

A continuación, se muestra el pseudocódigo del algoritmo colonia de hormigas:

```
Inicializar población y parámetros

MIENTRAS no se cumpla el criterio de terminación

         ConstruirSoluciones ()
         BusquedaLocal ()
         ActualizarFeromonas ()

FIN MIENTRAS
```

***Construcción de soluciones***: Construcción de la solución de cada hormiga. (Elecciones ruta)

***Búsqueda local***: Obtención de una solución vecina y comparación con esta con el fin de ver cuál es mejor.

***ActualizarFeromonas***: Evaporación de feromonas y posteriormente depósito de feromonas.

Por último, se muestra el pseudocódigo de la Búsqueda Tabú:

```
Inicializar población y parámetros

MIENTRAS no se cumpla el criterio de terminación

        Obtener vecindario de solución actual
        Obtener lista tabú
        Obtener conjunto de aspirantes

        Determinar nuevo vecindario reducido
        mediante las variables encontradas.

        Escoger la mejor solución del nuevo vecindario y
        guardarla si es mejor que la mejor solución actual.

        Actualizar la lista tabú.

FIN MIENTRAS
```

## B Código de la clase SoundRecorder

```
import javax.sound.sampled.*;
import java.io.*;

/**
 * A sample program is to demonstrate how to record sound in Java
 * author: www.codejava.net
 */
public class JavaSoundRecorder {
    // record duration, in milliseconds
    static final long RECORD_TIME = 60000;  // 1 minute

    // path of the wav file
    File wavFile = new File("E:/Test/RecordAudio.wav");

    // format of audio file
    AudioFileFormat.Type fileType = AudioFileFormat.Type.WAVE;

    // the line from which audio data is captured
    TargetDataLine line;

    /**
     * Defines an audio format
     */
    AudioFormat getAudioFormat() {
        float sampleRate = 16000;
        int sampleSizeInBits = 8;
        int channels = 2;
        boolean signed = true;
        boolean bigEndian = true;
        AudioFormat format = new AudioFormat(sampleRate,
sampleSizeInBits,
                                             channels, signed,
bigEndian);
        return format;
    }

    /**
     * Captures the sound and record into a WAV file
     */
    void start() {
        try {
            AudioFormat format = getAudioFormat();
            DataLine.Info info = new
DataLine.Info(TargetDataLine.class, format);

            // checks if system supports the data line
            if (!AudioSystem.isLineSupported(info)) {
                System.out.println("Line not supported");
                System.exit(0);
            }
            line = (TargetDataLine) AudioSystem.getLine(info);
            line.open(format);
            line.start();   // start capturing

            System.out.println("Start capturing...");

            AudioInputStream ais = new AudioInputStream(line);

            System.out.println("Start recording...");

            // start recording
            AudioSystem.write(ais, fileType, wavFile);
```

```
} catch (LineUnavailableException ex) {
            ex.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }

    /**
     * Closes the target data line to finish capturing and
recording
     */
    void finish() {
        line.stop();
        line.close();
        System.out.println("Finished");
    }

    /**
     * Entry to run the program
     */
    public static void main(String[] args) {
        final JavaSoundRecorder recorder = new
JavaSoundRecorder();

        // creates a new thread that waits for a specified
        // of time before stopping
        Thread stopper = new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(RECORD_TIME);
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
                recorder.finish();
            }
        });

        stopper.start();

        // start recording
        recorder.start();
    }
}
```

# C Diagrama de Gantt del proyecto

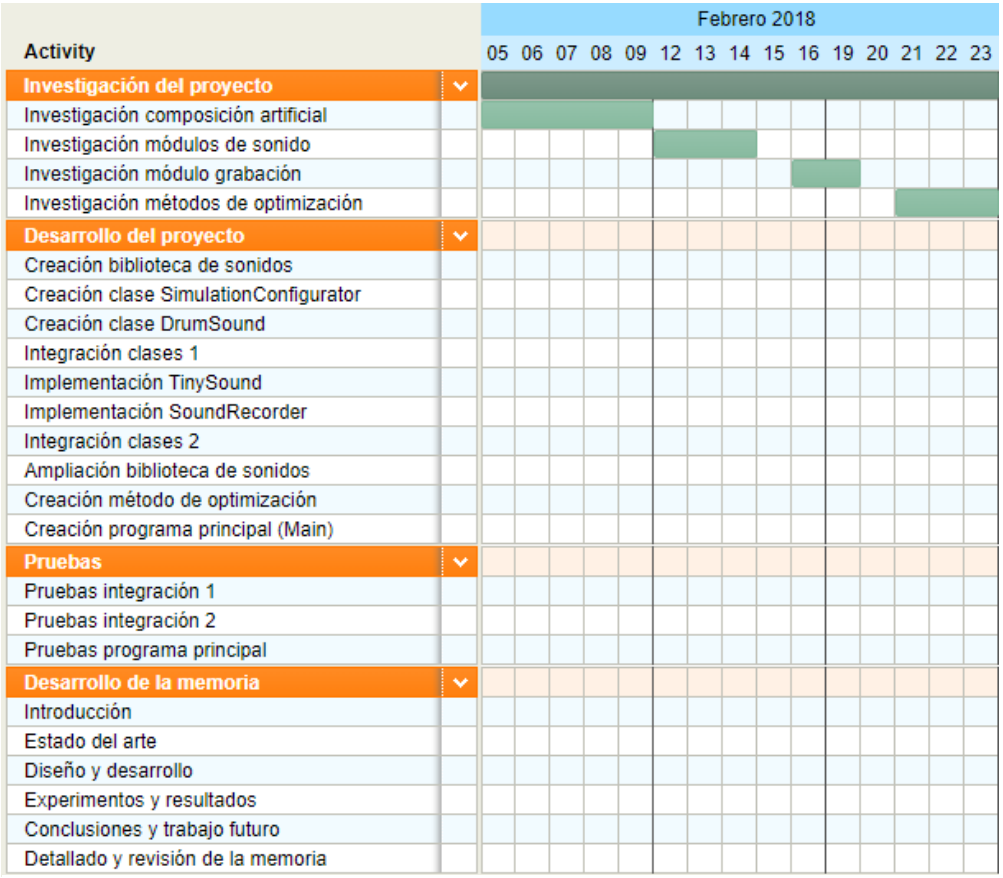



**Figura C-1 –** Diagrama Gantt del proyecto *(1)*

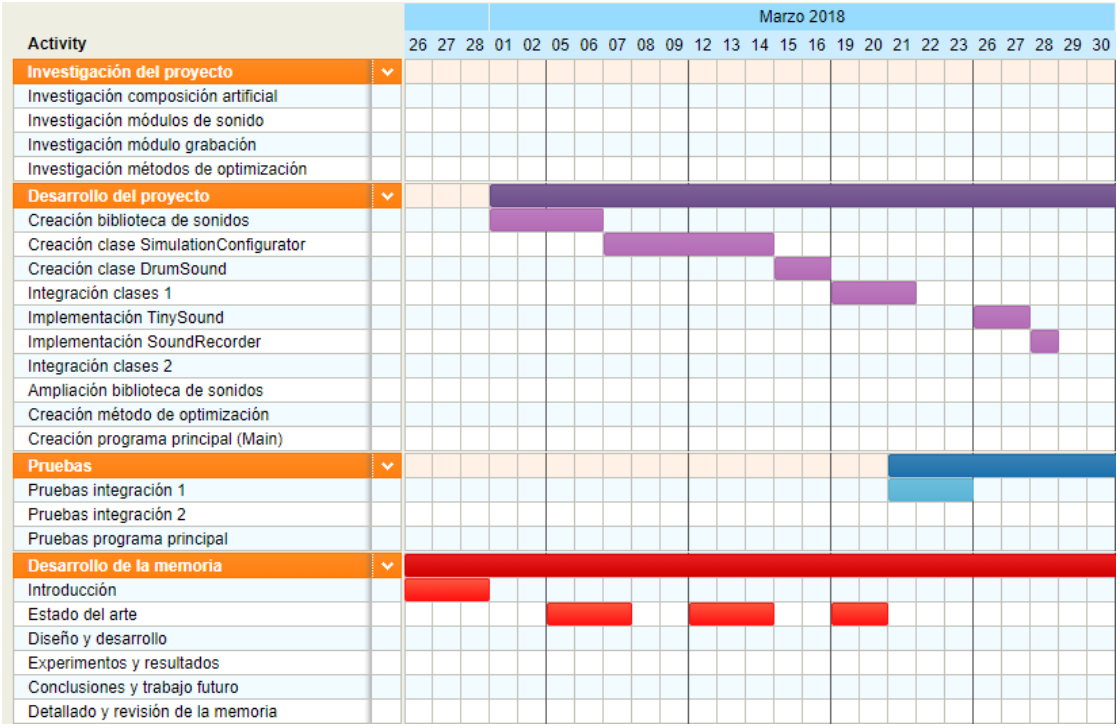



**Figura C-2 –** Diagrama Gantt del proyecto *(2)*

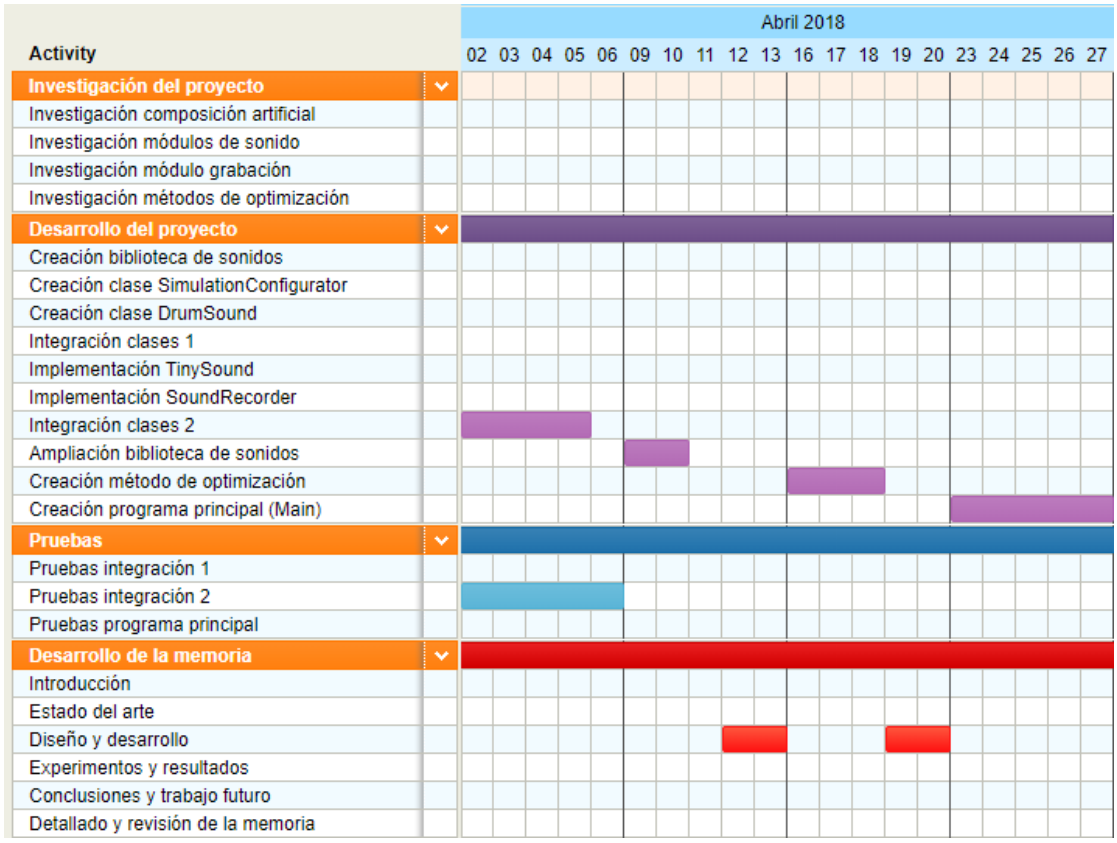



**Figura C-3** – Diagrama Gantt del proyecto *(3)*

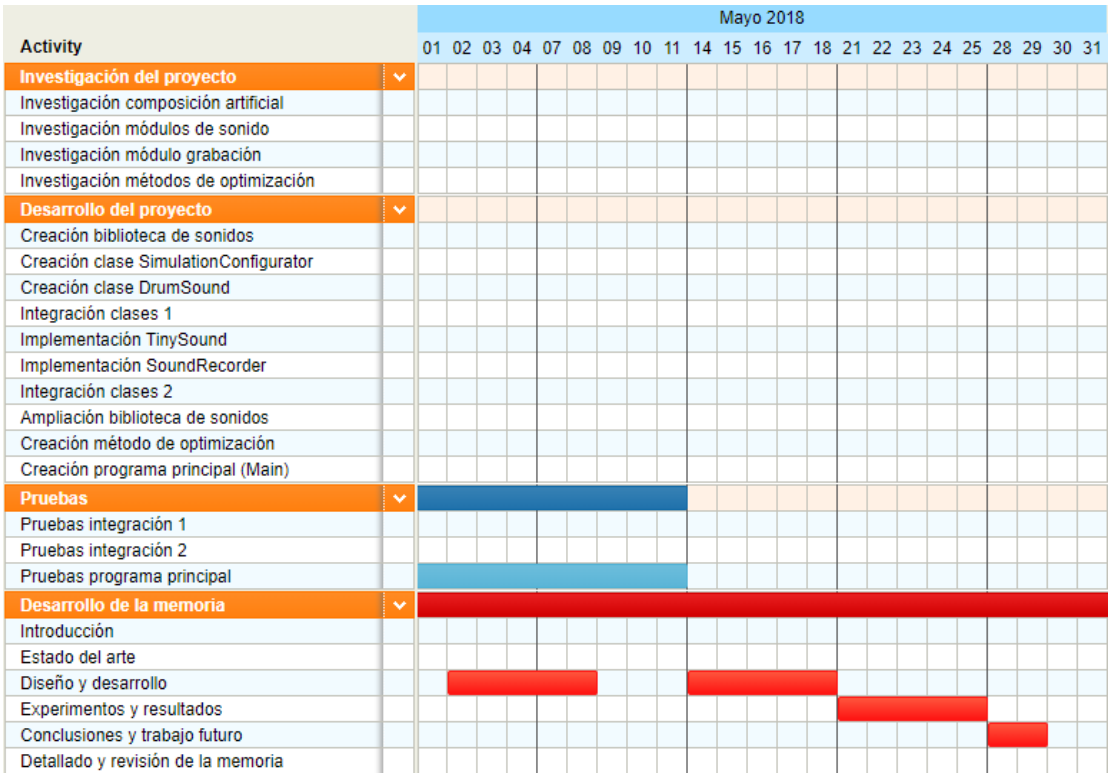



**Figura C-4** – Diagrama Gantt del proyecto *(4)*

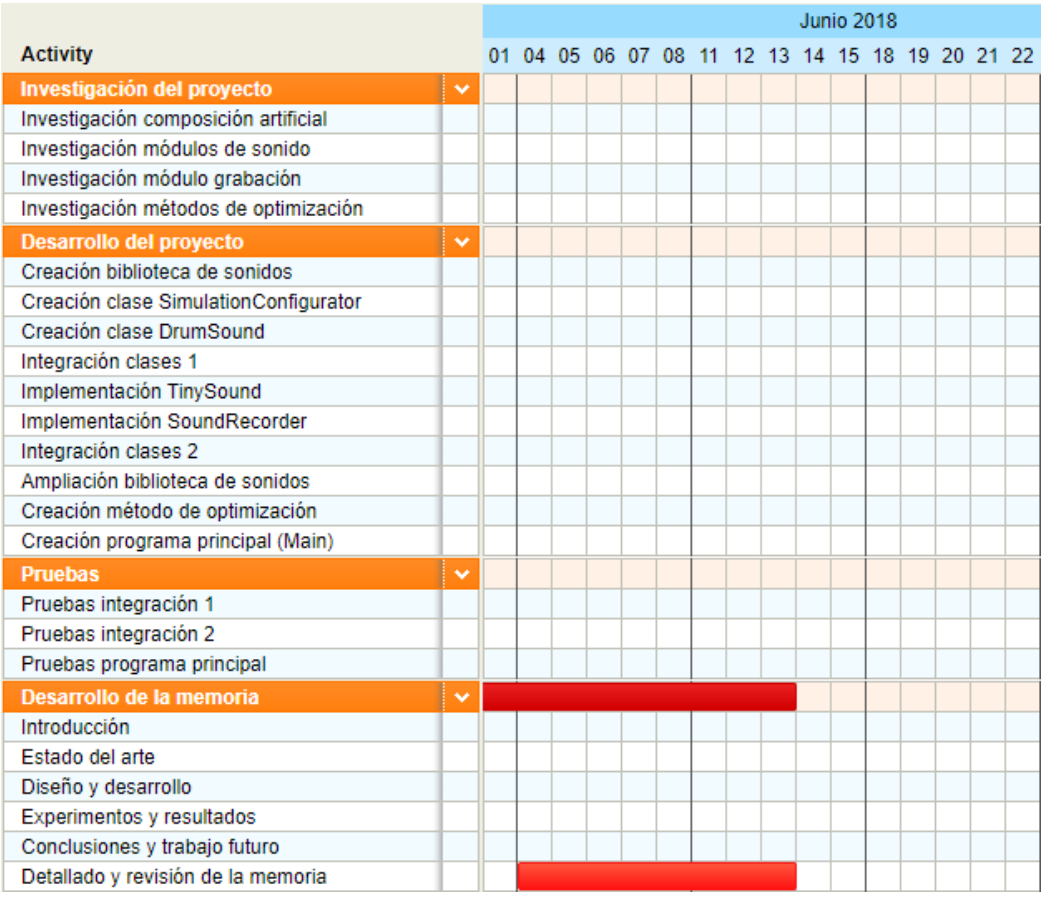



**Figura C-5** – Diagrama Gantt del proyecto *(5)*

## *D Resultados generación pseudoaleatoria (Prueba grande)*

| BPM | Número de sonidos | Kicks | Snares | Claps | HiHats Closed | HiHats Open | Cymbals | Percussion | Evaluación |
|---|---|---|---|---|---|---|---|---|---|
| **100** | 13 | 1 | 7 | 0 | 1 | 1 | 2 | 1 | 7.43 |
| **100** | 17 | 4 | 1 | 2 | 2 | 3 | 3 | 2 | 6.66 |
| **100** | 16 | 0 | 4 | 0 | 3 | 4 | 1 | 4 | 5.75 |
| **100** | 8 | 0 | 0 | 2 | 2 | 1 | 0 | 3 | 6.49 |
| **100** | 3 | 0 | 0 | 1 | 0 | 0 | 2 | 0 | 5.05 |
| **100** | 5 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 7.98 |
| **100** | 13 | 0 | 1 | 3 | 4 | 1 | 2 | 2 | 5.54 |
| **100** | 9 | 2 | 2 | 0 | 2 | 0 | 1 | 2 | 7.75 |
| **100** | 9 | 1 | 1 | 0 | 1 | 1 | 4 | 1 | 6.87 |
| **100** | 14 | 0 | 4 | 1 | 3 | 4 | 0 | 2 | 6.31 |
| **105** | 3 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 4.88 |
| **105** | 12 | 0 | 3 | 2 | 1 | 5 | 0 | 1 | 6.21 |
| **105** | 15 | 2 | 4 | 3 | 2 | 1 | 2 | 1 | 7.12 |
| **105** | 7 | 0 | 0 | 1 | 0 | 2 | 1 | 3 | 6.3 |
| **105** | 3 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 5.17 |
| **105** | 8 | 1 | 0 | 2 | 2 | 1 | 1 | 1 | 8.35 |
| **105** | 16 | 3 | 1 | 1 | 2 | 3 | 4 | 2 | 6.47 |
| **105** | 12 | 3 | 2 | 1 | 0 | 2 | 2 | 2 | 6.55 |
| **105** | 12 | 0 | 4 | 2 | 1 | 1 | 3 | 1 | 6.14 |
| **105** | 15 | 2 | 2 | 3 | 5 | 0 | 1 | 2 | 5.86 |
| **110** | 9 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 6.55 |
| **110** | 17 | 2 | 4 | 2 | 3 | 5 | 1 | 0 | 7.02 |
| **110** | 13 | 0 | 0 | 2 | 3 | 7 | 0 | 1 | 7.16 |
| **110** | 9 | 0 | 2 | 0 | 2 | 2 | 2 | 1 | 6.91 |
| **110** | 8 | 2 | 1 | 0 | 0 | 3 | 0 | 2 | 7.39 |
| **110** | 18 | 1 | 5 | 1 | 0 | 4 | 5 | 2 | 6.66 |
| **110** | 6 | 0 | 1 | 0 | 0 | 0 | 3 | 2 | 5.37 |
| **110** | 17 | 4 | 4 | 3 | 3 | 2 | 1 | 0 | 7.07 |
| **110** | 8 | 3 | 1 | 1 | 2 | 0 | 0 | 1 | 7.55 |
| **110** | 14 | 2 | 4 | 1 | 1 | 1 | 1 | 4 | 7.1 |
| **115** | 6 | 0 | 4 | 0 | 0 | 1 | 0 | 1 | 6.95 |
| **115** | 20 | 1 | 6 | 2 | 4 | 1 | 2 | 4 | 6.87 |
| **115** | 20 | 3 | 4 | 5 | 2 | 5 | 1 | 0 | 6.9 |
| **115** | 17 | 0 | 5 | 2 | 2 | 2 | 1 | 5 | 6.35 |
| **115** | 6 | 0 | 1 | 0 | 2 | 2 | 0 | 1 | 6.42 |
| **115** | 8 | 0 | 3 | 1 | 1 | 1 | 1 | 1 | 7.08 |
| **115** | 11 | 2 | 0 | 2 | 2 | 4 | 0 | 1 | 8 |
| **115** | 5 | 1 | 1 | 0 | 1 | 0 | 0 | 2 | 7.66 |
| **115** | 8 | 0 | 2 | 0 | 2 | 0 | 1 | 3 | 5.65 |
| **115** | 11 | 1 | 2 | 2 | 1 | 3 | 1 | 1 | 6.36 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **120** | 16 | 5 | 2 | 4 | 2 | 1 | 1 | 1 | 6.93 |
| **120** | 15 | 1 | 1 | 2 | 1 | 5 | 4 | 1 | 5.98 |
| **120** | 3 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 6.11 |
| **120** | 5 | 2 | 0 | 0 | 0 | 1 | 2 | 0 | 6.63 |
| **120** | 20 | 4 | 3 | 2 | 1 | 4 | 4 | 2 | 7.27 |
| **120** | 15 | 5 | 2 | 2 | 0 | 3 | 2 | 1 | 7.02 |
| **120** | 16 | 5 | 2 | 2 | 2 | 1 | 3 | 1 | 7.08 |
| **120** | 7 | 1 | 1 | 1 | 1 | 0 | 1 | 2 | 8.41 |
| **120** | 14 | 0 | 1 | 2 | 2 | 4 | 3 | 2 | 5.67 |
| **120** | 3 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 4.74 |
| **125** | 16 | 3 | 2 | 0 | 2 | 1 | 4 | 4 | 6.67 |
| **125** | 15 | 3 | 2 | 2 | 1 | 5 | 0 | 2 | 6.91 |
| **125** | 12 | 4 | 1 | 0 | 2 | 1 | 3 | 1 | 7.25 |
| **125** | 17 | 4 | 2 | 1 | 3 | 2 | 1 | 4 | 7.05 |
| **125** | 11 | 2 | 1 | 1 | 3 | 1 | 1 | 2 | 7.67 |
| **125** | 5 | 0 | 2 | 0 | 0 | 1 | 2 | 0 | 6.86 |
| **125** | 17 | 3 | 1 | 0 | 2 | 3 | 4 | 4 | 6.7 |
| **125** | 20 | 2 | 2 | 4 | 1 | 3 | 7 | 1 | 6.43 |
| **125** | 6 | 0 | 0 | 0 | 2 | 0 | 0 | 4 | 4.44 |
| **125** | 16 | 4 | 3 | 1 | 1 | 2 | 0 | 5 | 7.22 |
| **130** | 12 | 0 | 1 | 2 | 4 | 2 | 2 | 1 | 5.95 |
| **130** | 20 | 2 | 4 | 1 | 3 | 2 | 4 | 4 | 7.08 |
| **130** | 5 | 3 | 0 | 2 | 0 | 0 | 0 | 0 | 6.24 |
| **130** | 8 | 1 | 3 | 0 | 1 | 0 | 1 | 2 | 7.93 |
| **130** | 7 | 1 | 1 | 3 | 2 | 0 | 0 | 0 | 6.92 |
| **130** | 9 | 1 | 2 | 0 | 2 | 2 | 0 | 2 | 7.14 |
| **130** | 8 | 3 | 0 | 2 | 0 | 1 | 1 | 1 | 7.79 |
| **130** | 18 | 2 | 1 | 2 | 1 | 7 | 1 | 4 | 6.46 |
| **130** | 13 | 0 | 4 | 0 | 0 | 4 | 5 | 0 | 5.68 |
| **130** | 7 | 0 | 2 | 1 | 1 | 1 | 1 | 1 | 6.48 |
| **135** | 19 | 4 | 2 | 3 | 3 | 3 | 3 | 1 | 7.1 |
| **135** | 18 | 3 | 1 | 3 | 4 | 3 | 3 | 1 | 6.54 |
| **135** | 12 | 1 | 3 | 1 | 0 | 2 | 2 | 3 | 5.43 |
| **135** | 6 | 1 | 0 | 1 | 1 | 2 | 0 | 1 | 8.28 |
| **135** | 4 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 6.88 |
| **135** | 18 | 2 | 1 | 0 | 2 | 5 | 5 | 3 | 5.99 |
| **135** | 5 | 1 | 0 | 2 | 0 | 1 | 0 | 1 | 7.14 |
| **135** | 8 | 1 | 1 | 0 | 1 | 1 | 2 | 2 | 8.16 |
| **135** | 17 | 3 | 0 | 1 | 6 | 3 | 1 | 3 | 6.72 |
| **135** | 9 | 1 | 3 | 0 | 2 | 1 | 0 | 2 | 7.1 |
| **140** | 16 | 2 | 6 | 1 | 1 | 1 | 2 | 3 | 7.03 |
| **140** | 16 | 4 | 3 | 1 | 1 | 2 | 2 | 3 | 7.11 |
| **140** | 6 | 1 | 0 | 1 | 2 | 1 | 1 | 0 | 8.29 |
| **140** | 17 | 3 | 1 | 3 | 2 | 2 | 4 | 2 | 6.31 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **140** | 14 | 2 | 3 | 1 | 2 | 3 | 2 | 1 | 6.92 |
| **140** | 8 | 0 | 3 | 0 | 0 | 3 | 1 | 1 | 6.7 |
| **140** | 11 | 1 | 2 | 1 | 4 | 1 | 2 | 0 | 6.14 |
| **140** | 19 | 3 | 2 | 2 | 0 | 2 | 7 | 3 | 5.96 |
| **140** | 4 | 2 | 0 | 1 | 0 | 0 | 0 | 1 | 5.9 |
| **140** | 15 | 2 | 3 | 4 | 5 | 0 | 0 | 1 | 6.17 |
| **145** | 7 | 0 | 1 | 0 | 0 | 3 | 2 | 1 | 5.93 |
| **145** | 13 | 2 | 2 | 1 | 6 | 1 | 0 | 1 | 6.29 |
| **145** | 12 | 0 | 2 | 3 | 2 | 1 | 1 | 3 | 5.81 |
| **145** | 8 | 1 | 3 | 0 | 0 | 1 | 1 | 2 | 7.99 |
| **145** | 11 | 4 | 1 | 0 | 0 | 3 | 1 | 2 | 6.17 |
| **145** | 3 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 5.9 |
| **145** | 4 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 8.04 |
| **145** | 5 | 0 | 3 | 1 | 0 | 0 | 0 | 1 | 6.21 |
| **145** | 3 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 5.56 |
| **145** | 16 | 1 | 3 | 2 | 1 | 2 | 4 | 3 | 6.37 |
| **150** | 13 | 1 | 1 | 3 | 2 | 3 | 0 | 3 | 6.22 |
| **150** | 16 | 1 | 4 | 1 | 4 | 3 | 2 | 1 | 6.92 |
| **150** | 20 | 3 | 1 | 3 | 5 | 2 | 3 | 3 | 6.29 |
| **150** | 17 | 3 | 1 | 2 | 4 | 3 | 1 | 3 | 6.43 |
| **150** | 3 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 6.92 |
| **150** | 19 | 4 | 4 | 1 | 2 | 2 | 2 | 4 | 7 |
| **150** | 16 | 2 | 1 | 2 | 2 | 3 | 5 | 1 | 6.4 |
| **150** | 4 | 0 | 0 | 2 | 0 | 0 | 1 | 1 | 5.86 |
| **150** | 4 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 5.74 |
| **150** | 11 | 2 | 2 | 1 | 2 | 2 | 0 | 2 | 8.22 |
| **155** | 16 | 3 | 2 | 2 | 1 | 1 | 3 | 4 | 6.75 |
| **155** | 16 | 5 | 1 | 2 | 1 | 0 | 3 | 4 | 6.42 |
| **155** | 6 | 3 | 0 | 1 | 0 | 0 | 2 | 0 | 7.41 |
| **155** | 17 | 4 | 2 | 2 | 2 | 3 | 1 | 3 | 6.85 |
| **155** | 12 | 0 | 3 | 3 | 1 | 3 | 2 | 0 | 7.52 |
| **155** | 13 | 1 | 2 | 3 | 3 | 1 | 3 | 0 | 6.4 |
| **155** | 7 | 0 | 0 | 1 | 1 | 0 | 3 | 2 | 5.61 |
| **155** | 18 | 4 | 0 | 5 | 0 | 2 | 3 | 4 | 6.47 |
| **155** | 10 | 1 | 2 | 2 | 1 | 2 | 0 | 2 | 6.97 |
| **155** | 16 | 3 | 1 | 1 | 1 | 3 | 1 | 6 | 6.37 |
| **160** | 16 | 4 | 2 | 4 | 1 | 0 | 2 | 3 | 6.42 |
| **160** | 12 | 2 | 1 | 2 | 2 | 2 | 1 | 2 | 7.74 |
| **160** | 4 | 0 | 1 | 0 | 0 | 1 | 2 | 0 | 5.86 |
| **160** | 16 | 1 | 6 | 3 | 1 | 4 | 0 | 1 | 7.46 |
| **160** | 10 | 1 | 0 | 6 | 1 | 1 | 0 | 1 | 7.67 |
| **160** | 9 | 1 | 1 | 2 | 2 | 2 | 1 | 0 | 8.3 |
| **160** | 6 | 1 | 0 | 0 | 0 | 1 | 1 | 3 | 6.38 |
| **160** | 9 | 0 | 4 | 2 | 0 | 2 | 0 | 1 | 5.7 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **160** | 4 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 7.04 |
| **160** | 3 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 5.74 |
| **165** | 13 | 0 | 0 | 3 | 0 | 2 | 5 | 3 | 5.91 |
| **165** | 10 | 0 | 1 | 0 | 3 | 4 | 0 | 2 | 5.51 |
| **165** | 9 | 3 | 1 | 2 | 1 | 1 | 1 | 0 | 7.05 |
| **165** | 20 | 1 | 5 | 2 | 4 | 4 | 1 | 3 | 7.03 |
| **165** | 18 | 4 | 3 | 2 | 1 | 2 | 3 | 3 | 7.06 |
| **165** | 6 | 0 | 1 | 0 | 1 | 0 | 2 | 2 | 6.8 |
| **165** | 12 | 2 | 2 | 1 | 0 | 3 | 1 | 3 | 6.22 |
| **165** | 10 | 2 | 0 | 2 | 1 | 1 | 3 | 1 | 7.19 |
| **165** | 13 | 1 | 2 | 2 | 3 | 1 | 2 | 2 | 6.45 |
| **165** | 9 | 1 | 0 | 1 | 1 | 2 | 3 | 1 | 6.59 |
| **170** | 19 | 5 | 3 | 5 | 1 | 5 | 0 | 0 | 7.02 |
| **170** | 20 | 2 | 2 | 2 | 4 | 2 | 3 | 5 | 6.49 |
| **170** | 8 | 2 | 0 | 2 | 0 | 1 | 2 | 1 | 7.49 |
| **170** | 19 | 1 | 5 | 4 | 1 | 2 | 4 | 2 | 6.93 |
| **170** | 17 | 5 | 3 | 2 | 0 | 3 | 1 | 3 | 6.63 |
| **170** | 16 | 2 | 3 | 2 | 3 | 1 | 2 | 3 | 6.78 |
| **170** | 3 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 6.13 |
| **170** | 3 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 5.73 |
| **170** | 12 | 2 | 1 | 2 | 0 | 0 | 3 | 4 | 5.38 |
| **170** | **7** | **1** | **1** | **1** | **1** | **1** | **2** | **0** | **8.84** |
| **175** | 8 | 1 | 1 | 2 | 0 | 0 | 1 | 3 | 6.15 |
| **175** | 15 | 3 | 3 | 2 | 2 | 1 | 2 | 2 | 7.33 |
| **175** | 18 | 0 | 4 | 3 | 3 | 1 | 3 | 4 | 6.17 |
| **175** | 13 | 1 | 1 | 1 | 2 | 5 | 0 | 3 | 6.05 |
| **175** | 5 | 0 | 1 | 2 | 0 | 0 | 1 | 1 | 6.3 |
| **175** | 3 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 4.88 |
| **175** | 4 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 5.49 |
| **175** | 18 | 1 | 6 | 4 | 2 | 2 | 1 | 2 | 6.96 |
| **175** | 16 | 1 | 1 | 5 | 1 | 3 | 3 | 2 | 6.03 |
| **175** | 19 | 1 | 1 | 8 | 2 | 2 | 2 | 3 | 5.96 |
| **180** | 5 | 1 | 0 | 1 | 3 | 0 | 0 | 0 | 7.16 |
| **180** | 3 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 6.35 |
| **180** | 20 | 2 | 2 | 2 | 4 | 2 | 3 | 5 | 6.27 |
| **180** | 19 | 2 | 1 | 3 | 5 | 4 | 1 | 3 | 6.51 |
| **180** | 4 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 7.27 |
| **180** | 10 | 2 | 2 | 0 | 2 | 2 | 1 | 1 | 7.55 |
| **180** | 8 | 0 | 1 | 0 | 2 | 1 | 3 | 1 | 5.64 |
| **180** | 20 | 0 | 5 | 3 | 6 | 3 | 2 | 1 | 6.04 |
| **180** | 5 | 0 | 1 | 0 | 0 | 1 | 3 | 0 | 6.36 |
| **180** | 20 | 2 | 4 | 3 | 0 | 4 | 4 | 3 | 6.45 |
| **185** | 12 | 1 | 2 | 5 | 0 | 1 | 2 | 1 | 7.01 |
| **185** | 17 | 5 | 4 | 2 | 3 | 1 | 0 | 2 | 6.99 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **185** | 12 | 2 | 1 | 2 | 2 | 2 | 2 | 1 | 8.84 |
| **185** | 8 | 0 | 1 | 1 | 0 | 1 | 2 | 3 | 5.29 |
| **185** | 3 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 5.11 |
| **185** | 13 | 4 | 2 | 0 | 0 | 1 | 3 | 3 | 6.55 |
| **185** | 20 | 3 | 1 | 3 | 3 | 2 | 3 | 5 | 6.36 |
| **185** | 8 | 2 | 3 | 0 | 2 | 0 | 0 | 1 | 6.44 |
| **185** | 18 | 3 | 2 | 1 | 4 | 0 | 4 | 4 | 6.14 |
| **185** | 19 | 2 | 3 | 2 | 2 | 5 | 2 | 3 | 6.73 |
| **190** | 10 | 2 | 1 | 0 | 3 | 4 | 0 | 0 | 6.42 |
| **190** | 16 | 1 | 5 | 1 | 3 | 2 | 1 | 3 | 7.08 |
| **190** | 4 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 5.38 |
| **190** | 8 | 1 | 1 | 0 | 2 | 2 | 0 | 2 | 8.51 |
| **190** | 9 | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 7.3 |
| **190** | 9 | 1 | 1 | 2 | 1 | 0 | 3 | 1 | 7.26 |
| **190** | 5 | 2 | 0 | 1 | 0 | 1 | 1 | 0 | 7.13 |
| **190** | 16 | 0 | 5 | 0 | 1 | 5 | 2 | 3 | 6.62 |
| **190** | 18 | 3 | 2 | 3 | 4 | 2 | 0 | 4 | 6.48 |
| **190** | 17 | 3 | 3 | 2 | 2 | 4 | 2 | 1 | 7.01 |
| **195** | 9 | 1 | 0 | 1 | 1 | 1 | 3 | 2 | 7.54 |
| **195** | 5 | 0 | 0 | 2 | 0 | 0 | 1 | 2 | 6.66 |
| **195** | 12 | 4 | 4 | 1 | 0 | 2 | 0 | 1 | 6.84 |
| **195** | 18 | 2 | 5 | 3 | 1 | 4 | 1 | 2 | 7.01 |
| **195** | 10 | 3 | 0 | 2 | 1 | 3 | 1 | 0 | 6.8 |
| **195** | 3 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 5.75 |
| **195** | 8 | 0 | 4 | 0 | 2 | 2 | 0 | 0 | 7.86 |
| **195** | 3 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 4.8 |
| **195** | 10 | 0 | 3 | 1 | 2 | 0 | 2 | 2 | 5.73 |
| **195** | 10 | 0 | 2 | 1 | 2 | 1 | 1 | 3 | 5.71 |

**Tabla D.1 –** Resultado de 200 ejecuciones diferentes